

A STRUCTURED DOCUMENT MODEL

by

Clement Shin Tai-Yin , AP(HK), MSc, DIC

December 1988

A thesis submitted for the degree of Doctor of Philosophy
of the University of London

Department of Computing
Imperial College of Science & Technology
University of London
London SW7 2BZ

To my parents

ACKNOWLEDGEMENTS

I am most grateful to my supervisor Dr. Donald M. Monro for introducing me to the exciting field of document processing. His enthusiasm and prolonged involvement in the area have resulted in valuable guidance and inspiration during the past four years. Thanks are also due to Prof. Bruce McA. Sayers for his constant care and support for our group.

I am indebted to Eddie Kwok, Barry Sherlock and David Simpson for their many useful suggestions and ideas. I must also express my gratitude to the Croucher Foundation for their generous financial support.

Finally, and above all, I wish to thank my family for their spiritual as well as financial support.

ABSTRACT

A structured document model is defined in this thesis which achieves a more complete and consistent logical decomposition of the elementary objects which make up a document than has been reported previously. Existing hierarchical document models have treated complex objects like equations and figures as terminal elements which are not logically decomposed adequately. Consequently, the structured editing philosophy typically supports only the high-level organization of a document but terminates at these complex object levels. The architecture is intended to support the abstract design of a system in which structure editing would apply to all levels of the document hierarchy, from the chapter level down to complex but structured objects.

An interactive editor for mathematical expressions has been implemented as an example of a structured editor for complex objects. It is based on the syntactic definition of the object *equation* in the model and, with its template-driven and structured approach, has overcome some disadvantages of existing tools. Furthermore, the geometric and syntactic information embedded in the mathematical templates has enabled declarative specification, which has previously appeared at the document description level.

An investigation is presented of the generalization of logically structured diagrams in the document model. The proposed approach is based on the definition of *illustration class* for diagrams having a similar underlying logical structure, which share an *abstract model* and a *virtual editor* accordingly. The requirements

and difficulties that would be involved in adapting the model for specific diagrams are studied.

Tabular materials are treated as a generalized form of the relational data structure. The association of individual fields with an object type (such as equation, image, or text) would enable column-oriented operations which are specific to the respective entry types. Specific table instances would be compatible with an underlying database when the relational restrictions are enforced.

Finally, a font caching scheme which exploits the statistical properties of English text is proposed. Although not being part of the document model, it is an implementation issue in which some original ideas are presented. Simulations have shown better performance than strategies which are not based on English statistics.

STATEMENT OF ORIGINALITY

The following constitute the original contribution of the work presented in this thesis.

1. The definition of a document model with a more comprehensive logical organization of elementary document objects than previously achieved.
2. The implementation of a structured interactive editor for mathematical expressions which is based on an explicitly defined logical model.
3. The proposal and study of a model intended to generalize a subset of logically oriented diagrams.
4. The proposal and evaluation of a font caching scheme for interactive applications, which is based on the statistical behavior of English text.

CONTENTS

Title Page	1
Dedications	2
Acknowledgements	3
Abstract	4
Statement of Originality	6
Contents	7
List of Figures	13
List of Tables	15
 Chapter 1 INTRODUCTION	 16
1.1 DOCUMENT PROCESSING : Then and Now	16
1.2 OBJECTIVES AND APPROACH	19
1.3 ACHIEVEMENTS	21
1.4 OUTLINE OF THESIS	23
 Chapter 2 DOCUMENT PROCESSING BACKGROUND	 25
2.1 DOCUMENT MODELING	25
2.1.1 Structural Model	26
2.1.2 Processing Model	28
2.1.3 Existing Document Models	30
2.1.3.1 Office Document Architecture	30
2.1.3.2 Standard Generalized Markup Language	32
2.1.3.3 Interscript	33
2.1.3.4 Document Content Architecture	34
2.1.3.5 Kimura & Shaw's Model	34
2.1.3.6 Peels et al's Model	35
2.1.3.7 Quint & Vatton's Model - Grif	36
2.1.3.8 Furuta's Model	37
2.2 RELEVANT CONCEPTS	38
2.2.1 The Edit-Format Cycle	38
2.2.2 Content-Style Separation	40
2.2.3 Structure Editing	42
2.2.4 Non-linear Text Models	44
2.3 EXISTING SYSTEMS	46

2.3.1 Influential Systems	46	
2.3.1.1 T _E X	46	
2.3.1.2 Scribe	47	
2.3.1.3 Star 8010	48	
2.3.2 Commercial Developments	49	
2.3.3 Current Research Emphases	51	
2.3.3.1 Batch-or-WYSIWYG Dilemma	51	
2.3.3.2 Structured and Integrated Systems	54	
2.4 CONCLUSIONS		55
 Chapter 3 OVERALL DOCUMENT ARCHITECTURE		57
3.1 INTRODUCTION		57
3.1.1 Scope and Emphases	57	
3.1.2 An Overview	59	
3.1.3 Notation	61	
3.2 LOGICAL MODEL		62
3.2.1 Fundamental Structure	62	
3.2.1.1 Composite Objects	63	
3.2.1.2 Elementary Objects	64	
3.2.1.3 Non-hierarchical Links	66	
3.2.2 Auxiliary Structure	67	
3.2.2.1 Writing Support	68	
3.2.2.2 Reading Support	69	
3.2.3 Instantiation Rules	69	
3.2.3.1 Elementary Object Nesting	70	
3.2.3.2 Inter-object Links	71	
3.3 PHYSICAL MODEL		71
3.3.1 Elementary Areas	72	
3.3.2 Composite Areas	73	
3.4 PROCESSING MODEL		73
3.4.1 Editing Activities	74	
3.4.1.1 Global Structure Manipulation	74	
3.4.1.2 Modification of Architecture	76	
3.4.2 Formatting Activities	77	
3.4.2.1 Elementary Object Construction	78	
3.4.2.2 Composite Object Construction	78	
3.4.2.3 Page Composition	78	
3.5 AN ABSTRACT SYSTEM		80
3.5.1 Integrated and Structured Editing	80	
3.5.2 Window-based Activities	82	
3.6 CONCLUSIONS		85

Chapter 4	AN INTERACTIVE EQUATION EDITOR	86
4.1	INTRODUCTION	86
4.1.1	Problems	86
4.1.2	Existing Equation Tools	88
4.1.2.1	Language-based Tools	89
4.1.2.2	Direct-manipulation Tools	92
4.1.2.3	Computation-oriented Tools	93
4.2	AN ARCHITECTURE FOR EQUATIONS	95
4.2.1	Logical Model	95
4.2.2	A Box-binding Model	98
4.2.3	Formatting Rules	104
4.3	THE EDITOR/FORMATTER	106
4.3.1	Functional Blocks	107
4.3.2	User Interface	108
4.3.2.1	Cursor Movements	109
4.3.2.2	Interaction Requirements	109
4.3.3	A Structured Approach	110
4.3.3.1	Mathematical Templates	112
4.3.3.2	Structured Specification	114
4.3.4	Formatting Issues	115
4.3.4.1	Storage Formats	116
4.3.4.2	The Formatting Process	118
4.3.5	Overall Evaluation	119
4.3.5.1	The Model	120
4.3.5.2	The Editor	122
4.4	CONCLUSIONS	125
Chapter 5	THE ELEMENTARY TABULAR OBJECT	127
5.1	INTRODUCTION	127
5.1.1	Definitions	127
5.1.2	Relational Data Model	130
5.1.2.1	Structural Components	130
5.1.2.2	Manipulative Components	131
5.1.3	Some Relevant Tools	132
5.1.3.1	TBL/Troff	132
5.1.3.2	TABLE	134
5.1.3.3	Star	135
5.2	AN ARCHITECTURE FOR TABLES	135
5.2.1	Structural Components	136
5.2.1.1	The Logical Model	136
5.2.1.2	The Physical Model	141
5.2.2	Processing Components	145
5.2.2.1	Formatting Rules	145

5.2.2.2 Structure Manipulation	146	
5.3 AN ABSTRACT EDITOR		147
5.3.1 User Interface		148
5.3.1.1 Structure Definition	148	
5.3.1.2 Data Manipulation	149	
5.3.2 Formatting Issues		150
5.4 CONCLUSIONS		152
 Chapter 6 A STRUCTURED DIAGRAM MODEL		 153
6.1 INTRODUCTION		153
6.1.1 General Drawing Tools		154
6.1.1.1 Manual Systems	154	
6.1.1.2 Constraint-based Systems	155	
6.1.2 Specialized Graphics Systems		156
6.1.3 Objectives		157
6.2 A GENERALIZED MODEL		158
6.2.1 Diagram Classes		159
6.2.1.1 Tree-based Diagrams	159	
6.2.1.2 Network-based Diagrams	161	
6.2.1.3 Data-driven Diagrams	162	
6.2.2 Virtual Editing Functions		162
6.2.2.1 Tree Manipulation	163	
6.2.2.2 Graph Manipulation	164	
6.3 ON ADAPTING THE MODEL		165
6.3.1 Diagram Syntax		165
6.3.2 Physical Model		166
6.3.3 User Interface		167
6.3.4 Layout Procedures		167
6.4 CONCLUSIONS		169
 Chapter 7 ASSESSMENT OF THE UNIFIED ARCHITECTURE		 171
7.1 REVIEW OF THE STRUCTURED FORMALISM		171
7.2 EVALUATION OF THE MODEL		173
7.2.1 The Equation Object		174
7.2.1.1 Mathematical Structure	174	
7.2.1.2 Equation Binding Model	176	
7.2.1.3 Structured Equation Editing	177	
7.2.1.4 Declarative Equation Layout	184	
7.2.2 The Tabular Object		186
7.2.2.1 Logical Table Structure	186	
7.2.2.2 Physical Array Structure	187	

7.2.2.3 Structured Table Manipulation	188	
7.2.2.4 Automatic Table Generation	188	
7.2.3 Structured Diagrams		189
7.2.3.1 Motivation Revisited	190	
7.2.3.2 Logically oriented Diagrams	191	
7.2.3.3 Physical Diagram Structure	192	
7.2.3.4 Structured Diagram Manipulation	193	
7.2.3.5 Automatic Diagram Layout	193	
7.2.4 Composite Objects		195
7.2.4.1 Global Document Structure	195	
7.2.4.2 Page Layout Model	195	
7.2.4.3 Structured Document Manipulation	196	
7.2.4.4 Page-formatting Rules	197	
7.2.5 Summary of Appraisal		197
7.2.6 Limitations of the Unified Framework		198
7.3 FUTURE DIRECTIONS		202
7.3.1 Revising the Equation Editor		202
7.3.1.1 Alternative Specification Methods	202	
7.3.1.2 Automatic Symbolic Substitution	203	
7.3.1.3 Layout Control Mechanism	204	
7.3.1.4 Postprocessing by Batch Formatter	205	
7.3.1.5 Interface to Symbolic Packages	206	
7.3.1.6 Reconstruction of Mathematical Structure	207	
7.3.2 An Integrated but Hybrid Framework		208
7.3.2.1 An Overview	208	
7.3.2.2 Non-dominating Logical Structure	210	
7.3.2.3 Multiple Modes of Editing	211	
7.3.2.4 Alternative Formatting Mechanism	212	
7.5 CONCLUSIONS		212
 Chapter 8 CONCLUSIONS		 214
8.1 SUMMARY OF ACHIEVEMENTS		214
8.2 STRUCTURE versus FLEXIBILITY		216
 References		 218

Appendix A	Overall Document Architecture	233
Appendix B	Architecture for Equations	237
B.1	Logical Model	237
B.2	Physical Model	244
B.3	Formatting Rules	246
Appendix C	The Equation Editor	252
C.1	List of Commands	252
C.2	Implemented Templates	257
Appendix D	Architecture for Tables	261
D.1	Logical Model	261
D.2	Formatting Rules	262
Appendix E	Typographical Glossary and Acronyms	263
Appendix F	Statistical Font Caching	265
F.1	Introduction	265
F.2	Statistics of English	266
F.3	The Caching Scheme	268
F.3.1	An Overview	268
F.3.2	Initial Font Work Set	269
F.3.3	Font Faults	271
F.4	Cache Simulations	274
F.4.1	The Experiments	275
F.4.2	Evaluation Criteria	276
F.4.3	Discussions	278
F.5	Conclusions	283

LIST OF FIGURES

Figure 1-1	Developments in document processing systems	18
Figure 2-1	Major components in a document model	28
Figure 3-1	The global structure of a document	60
Figure 3-2	Overall structural model and related activities	79
Figure 3-4	Supervising and dispatched editors	81
Figure 3-5	A example illustration	82
Figure 3-5	A demonstrative scene in model editing	84
Figure 4-1	The logical composition of a summation	96
Figure 4-2	Dual representation of a matrix	97
Figure 4-3	Physical composition of a summation	99
Figure 4-4	Terminology used in setting equations	100
Figure 4-5	Some box-binding modes	101
Figure 4-6	Three options in vertical binding	102
Figure 4-7	Two formatting rules for summation	103
Figure 4-8	Logical-to-physical transformation	105
Figure 4-9	Functional blocks of the equation editor	106
Figure 4-10	Some system-provided mathematical templates	112
Figure 4-11	A user-defined template being expanded	112
Figure 4-12	An initial editing session	113
Figure 4-13	Subsequent manipulation	115
Figure 4-14	Comparing with EasyT _E X	122
Figure 4-15	Entering a matrix	124
Figure 5-1	Relationship among a base table, a view and a form	129
Figure 5-2	An example table	132
Figure 5-3	Typical forms of a table	137
Figure 5-4	Record-dominant representation	138
Figure 5-5	Field-dominant representation	139
Figure 5-6	Scaling and alignment in composing sub-tables	142
Figure 5-7	The physical array structure of a table	144
Figure 5-8	Structure definition for a table	149

Figure 6-1	Applying constraints to a pattern	155
Figure 7-1	The Structured Approach	172
Figure F-1	The font cache	270
Figure F-2	Trigram probabilities illustrated by the sequence 'pso'	273
Figure F-3	Inter-fault distributions	279
Figure F-4	Threshold versus inter-fault distributions for STAT-S	281
Figure F-5	Threshold versus MDBF and bad caches for STAT-S	282

LIST OF TABLES

Table 2-1	Cross references for notations in declarative systems	42
Table 3-1	Nesting rules for elementary objects	70
Table 3-2	Connection rules for inter-object links	71
Table F-1	Results of simulations for various methods	278
Table F-2	Results of STAT-S using various thresholds	280

Chapter 1

Introduction

In this chapter the development of computer aided document processing is reviewed. Although one of the oldest areas of computer applications, document processing is still an active area of research. This leads to a discussion of the purpose, approach and achievements of this project. Finally, an overview of this thesis will be presented.

1.1 DOCUMENT PROCESSING : Then and Now

The application of the computer to composition and printing dates back to the 1960's. These first generation formatting systems were based on a low-level source language for document specification, and were closely bound to a particular printer or typesetter (that is, without the notion of device independent formatter output). Examples include the Runoff [Saltzer 1965] and the Format [Berns 1968] programs.

The second generation of text formatters appeared in the early 1970's. Although still batch-oriented, these systems were more flexible and generated output format which was independent of the printing devices. Furthermore, the content and format of a document were distinguished through macros which separated the specification of a document's structural elements and the corresponding typographical actions required. A typical example is the Troff formatter and the associated preprocessors and macros facilities [Ossanna 1977], which are still widely used in the UNIX environment. The development of batch formatters slowed down towards the late 1970's, although at least two notable

batch systems appeared around that time, namely, the Scribe [Reid 1980b] and the T_EX [Knuth 1979] formatting system.

A new class of interactive document systems, colloquially known as WYSIWYG (what you see is what you get) editors, has been responsible for the 'decline' of batch systems. One of the earliest interactive systems was Xerox's experimental editor Bravo developed in 1975 [Lampson 1978]. As personal workstations became more powerful, and as research on interactive computer systems matured, efforts have been diverted over the last decade from the batch to the WYSIWYG rationale. This has eventually resulted in the development of so-called *desktop publishing* environments in recent years. Figure 1-1 gives an overall picture of this transition. It must however be emphasized that for some good reasons batch systems are still widely used. In fact, both batch and interactive systems have their own merits and one of the current interests is in providing some sort of compromise philosophy [Chen & Harrison 1988] (§2.3.3.1).

Accompanying this 'batch to interactive' transition was the trend towards integration of text, graphics and images, exploiting raster graphics technology which has enabled a unified treatment of textual and graphical elements. In fact, one definition of the word 'text' (that is, "all information for human comprehension that is intended for presentation in a two-dimensional form" [ISO 1984]) has made the phrase 'integrated text and graphics' redundant. However, the word 'text' will carry its conventional meaning throughout this thesis, that is, with graphical objects or the like excluded.

Apart from static graphics, recent interest has been shown in integrating dynamic objects as well, such as video animation and digitized voice, resulting in so-called *compound* or *multimedia* documents (§2.2.4). Although beyond the focus

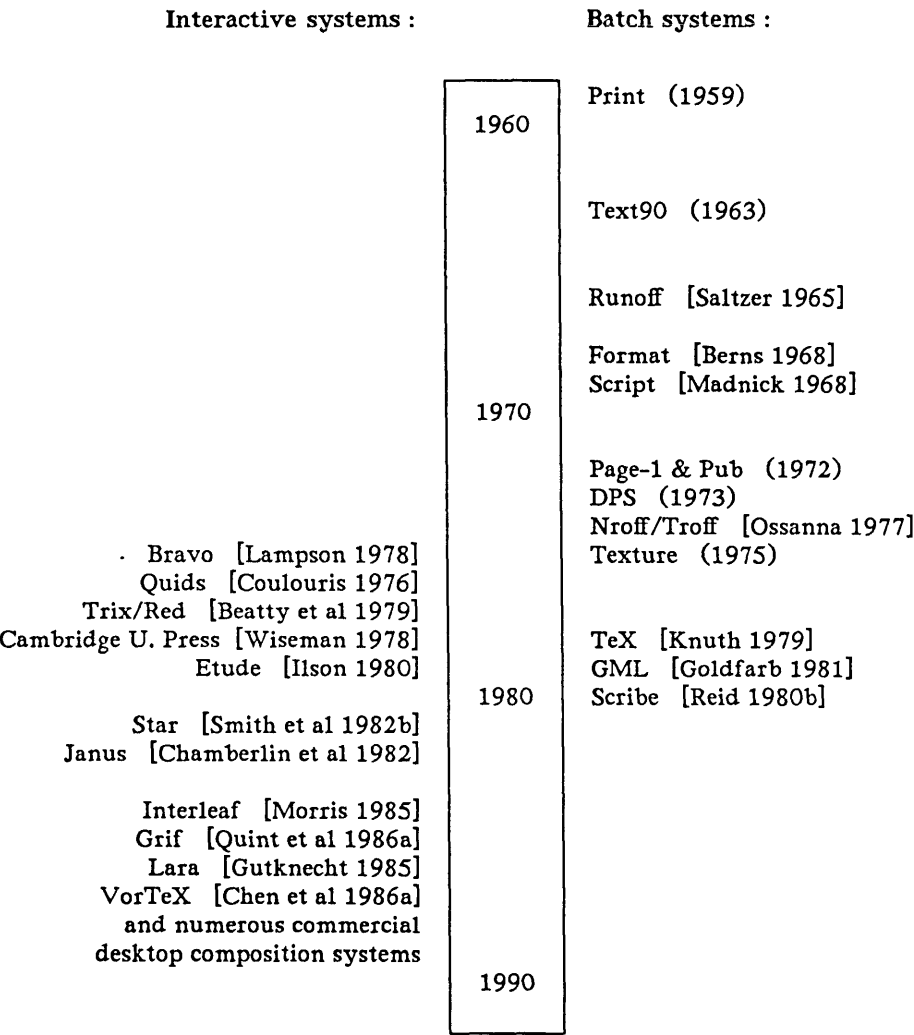


Figure 1-1. Developments in document processing systems

of this thesis, this indicates how document processing has evolved over the last few decades and remained as an area deserving further investigation.

1.2 OBJECTIVES AND APPROACH

Most interactive WYSIWYG systems perform instant formatting and pagination, and support flexible integration of text and graphics. These systems offer an attractive alternative to their batch counterparts, but are powerful basically within the context of *document preparation*.

Before we declare our goals, let us first examine a ‘knowledgeable’ document processing system which is here defined as one that understands document objects not just typographically, but at the syntactic and semantic levels as well. Such a system would thus be aware of the mathematical structure of an equation instead of treating it merely as some arbitrarily positioned and scaled characters. Likewise, it would understand, say, a state diagram rather than viewing it as some randomly connected graphical objects. There are three important implications of a system with such knowledge.

- A. **Structure editing.** The system’s *syntactic* knowledge of objects would enable structure editing, which has the well known features that a user can manipulate objects in logical terms rather than composing their physical form procedurally. Furthermore, provision of object templates would help to ensure creation of structurally valid objects. As far as document preparation is concerned, this is the most direct benefit of the system’s knowledge. Further discussion on structure editing will be found in Section 2.2.3.
- B. **Specialized assistance.** Based on its *semantic* knowledge of specific objects, the system could provide specialized assistance which, although not strictly essential, would be desirable in the document preparation process. Furthermore, authors are increasingly involved in documentation work, and often need more specific assistance than ordinary keyboard operators would.

One practical situation would be an equation editor providing computational or analytical power to some extent.

- C. **External collaboration.** Since the logical structure of objects would be maintained, the system could collaborate more directly with external applications which analyze and manipulate these objects. A user might, for instance, invoke an algebraic package while editing an equation, a circuit analysis program while preparing a circuit diagram, or a compiler while editing a program (as in a language sensitive program editor).

Our purpose here is to build a document model which incorporates structural (or syntactic) knowledge of document objects. The model is intended to serve the abstract design of an integrated and structured system which would maintain a logical view of the objects it manipulates. The model will not encompass any semantic knowledge, as it would not be appropriate for a document system, being one particular graphical application, to have a complete semantic model for objects being handled by others. With reference to the 'knowledgeable' system described above, the abstract system we have aimed at would therefore fully support structure editing, but would not provide any specialized assistance. The latter could, however, be achieved through collaboration with external specialized tools which have a better semantic knowledge of the object in question.

In building the model, special attention has been paid to two aspects. First, existing document models have treated complex objects like equations and figures as terminal elements which are not further decomposed adequately (as will be discussed in §2.1.3). Consequently, modern interactive systems typically only support structural manipulation of high-level document constructs ranging from paragraphs to chapters. To bridge this discontinuity, emphasis is here put on a

more complete logical organization of complex objects in order to support structure editing at all levels, from swapping two chapters to editing an equation.

Second, apart from physical integration of objects in a document, we are also concerned about integration of the *processes* that produce these objects. In a batch system, objects are typically handled by dedicated preprocessors, which often constrains the flexibility of nesting among various object types (as is in UNIX). In an interactive system, objects are often prepared with separate packages and then imported to a main composition process for integration. This has undoubtedly resulted in high flexibility, but still it would be desirable to have an integrated and structured process for handling various object types. Another emphasis in defining the model here is, therefore, to enable editors specialized for various objects to work in an integrated and coherent manner.

1.3 ACHIEVEMENTS

The major achievements in this work are as follows.

A. The design of a structured document model.

A logical document model is defined which aims at overcoming the inadequacy of existing architectures, and therefore emphasizes a more comprehensive structuring and integration of elementary document objects. Three types of objects which commonly appear in scientific documents (namely, equations, tables, and structured diagrams) are studied in detail. The model also serves the abstract design of an integrated system in which a crew of specialized object editors would be dispatched by a global document

structure editor.

B. The implementation of a WYSIWYG equation editor.

An interactive equation editor is implemented as an example of a specialized structure editor, chosen for its usefulness and rarity. Existing structured editors typically lack a comprehensive logical view of mathematical expressions. The editor here is based on a model defined as part of the document architecture, and its knowledge of mathematical syntax is incorporated in the form of mathematical templates. This template-driven and structured approach to manipulate equations has overcome some disadvantages of existing tools. Furthermore, geometric and syntactic information embedded in the templates have enabled a declarative approach to specifying mathematical expressions.

C. The proposal of a generalized model for structured diagrams.

A generalized model is proposed to accommodate a subset of logically oriented diagrams, which are not included in existing document models (due to the diversity involved). The motivation is that a structured document system should provide diagram-specific syntax direction for at least some common logically oriented diagrams. Three illustration classes have been defined for diagrams having a similar underlying structure. Diagrams belonging to a certain class share an abstract model and a corresponding virtual editor accordingly. We also study the difficulties that would arise when adapting the model for a specific type of diagrams.

D. The proposal of a statistical font caching scheme.

In a typical WYSIWYG document system, there is a necessary trade-off between the quality of screen simulation of typefaces and the speed of their generation. The latency associated with character generation can be reduced by caching characters in the available memory. A caching scheme that exploits the statistical nature of English text is proposed for such interactive environments. Simulations have demonstrated better performance than memory management schemes that do not exploit the statistical properties of English.

1.4 OUTLINE OF THESIS

The following gives a summary of the rest of this thesis.

Chapter 2 provides the reader with the necessary background on document processing. This includes some important concepts and a review of notable systems and recent developments.

Chapter 3 presents the design of the overall document model and a discussion on an abstract integrated system. The detailed architecture of three common document objects are dealt with in the following chapters.

Chapter 4 describes a model for mathematical expressions and the implementation of an interactive editor which is based on the model. An evaluation is presented comparing the model and the editor itself to existing counterparts.

Chapter 5 presents a model for tables. The logical component is based on the relational data model which has been generalized accordingly. The abstract design

of a structured editor for the object is presented.

Chapter 6 proposes a generalized model for a subset of logically structured diagrams. The procedures and difficulties involved in adapting the model for specific diagrams are studied.

Chapter 7 presents a critical study of the unified approach to formalizing the structural and functional characteristics of document objects, which has been adopted in the definition of our document model.

Chapter 8 is a concluding chapter that revisits our goals and achievements. Furthermore, radical and philosophical changes to the unified framework are suggested as some possible future directions for this project.

Chapter 2

Document Processing Background

This chapter summarizes previous work on document preparation that is relevant to our project. As the main theme of our work is on document modeling, the first section is dedicated to this. Then follows a section on other relevant concepts and issues in document processing. Finally, some important systems and recent research developments will be discussed.

2.1 DOCUMENT MODELING

Many text processing systems have been built successfully without assuming any document architecture as a unified framework. On the other hand, document systems may be designed on the basis of an explicitly defined model describing the structure and relations among constituent document objects. The work presented in this thesis aims at supporting the architectural approach to designing an integrated document system. There are two fundamental notions underlying a document architecture. The *structural model* represents the system's view of the document as a static structure, while the *processing model* specifies operations and activities that are performed on this structure. We first look at these two concepts in more detail, and then introduce several existing document models.

2.1.1 Structural Model

The structure of a document can be considered from two viewpoints, namely, the logical and the physical architecture.

- A. **Logical Model.** The logical (or abstract) model defines abstract elements appearing in a document and the relationship among these objects. It reflects how well an implementation understands the logical implications of individual document components. A simple text editor may view a piece of text as a simple string of characters without knowledge of words or sentences. Others may model a document to include high-level elements like paragraphs or sections. Specialized systems may understand specific structures like programs or data flow diagrams. A character-based editor and a language-sensitive editor may yield the same end product — a program, but the latter provides structure editing and syntax direction due to its knowledge of the language's syntax, and can communicate more closely with a compiler since the logical (or parsed) form of a program is normally maintained. A comprehensive logical model is essential for both declarative and structure editing in which logical elements must be identified (see §2.2.2 and §2.2.3).
- B. **Physical Model.** The physical (also known as layout or concrete) model defines components composing the external appearance of a document and the relationship among them. In a character-oriented device, the model may simply accommodate fixed-size character boxes. Proportional-spacing systems would allow arbitrary positioning of lines as well as characters. Objects other than ordinary text (like equations or figures) may require entirely different models since linear binding and text-only primitives will not suffice.

In applications like magazine layout, a page may be modeled as overlapping areas of arbitrary shapes, each being further structured according to its content. The physical model of a document defines application specific constructs and binding methods, but may be built upon lower level objects such as primitives supported in a general graphics system.

Two properties concerning the logical structure of a document are of organizational importance. The first is the *hierarchical* nature. Objects like chapter, section and paragraph imply a natural order of ranking. Many implementations have modeled a document as a tree structure, although provision for non-hierarchical relations is common. Examples include NLS [Engelbart et al 1968], XS-1 [Beretta et al 1982], Xerox's Tioga editor [Teitelman 1983], Walker's editor [Walker 1981], the Scribe formatter [Reid 1980b], and a more general tree-structure editor SDS [Fraser 1981].

The second property of a document's logical structure is the *class-instance* relationship. Every document element is associated with a certain class of objects having a similar structural and functional properties. For instance, all sections have similar composition (i.e. sub-classes) and require similar formatting actions (i.e. methods). This has led to object-oriented implementations such as Xerox's Star [Smith et al 1982b], the Tioga Editor [Teitelman 1983] and the PEN editor [Allen 1981], as well as document model specifications such as the standardized office document architecture ODA [ECMA 1985] and another formalism [Kimura & Shaw 1984]. The class-instance property also implies the use of formal grammar as the notation of document structure. Examples include Furuta's system [Furuta 1986], the COBATEF system [Peels et al 1985], the Grif integrated system [Quint et al 1986a], and the standard generalized markup language SGML [ISO 1986].

The Scribe [Reid 1980b] and T_EX [Knuth 1984] formatters are approximately at two extremes regarding their structural document models. While Scribe emphasizes the logical structure of a document but has a relatively casual physical model, T_EX possesses a sophisticated layout model, but almost totally ignores the logical aspects of a document. L^AT_EX, a macros package for T_EX which adds Scribe-oriented constructs to it, has provided a means of achieving both typesetting quality and declarative specification which are emphasized by the two systems respectively [Lamport 1986].

2.1.2 Processing Model

Four fundamental activities operate on the previously described structural model [Kimura & Shaw 1984]. The overall picture of a document model with its major structural and processing components is illustrated in Figure 2-1.

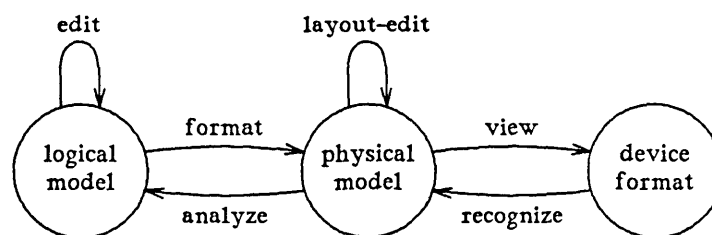


Figure 2-1. Major components in a document model

- A. **Editing.** This is the usual process of maintaining the logical structure of a document, in which the previous structure is mapped to a new instance. Typical operations include deleting, inserting or copying of words, sentences, paragraphs or sections (provided such objects exist in the structural model).

Alternatively, the physical structure of a document may be modified directly instead, where it is often referred to as *layout editing*. This occurs in applications for physically oriented documents where a user needs direct control over the page layout, such as positioning of floating objects.

An interactive system which supports both editing modes would allow a user to manipulate logical as well as physical objects, such as removing either a sentence or a formatted line. The set of possible editing operations depends directly on the logical model maintained by a system; it is this syntactic knowledge which structure editing basically exploits (§2.2.3).

- B. **Formatting.** This is the process of generating the physical form of a document from its logical structure. Logical constructs like section, paragraph and sentence are mapped to the physical counterparts like page, block and line. This generally involves tasks like hyphenation and justification, determining page breaks, positioning floating illustrations, or generating running headers. Formatters for specialized applications have knowledge of physical composition of the specific objects they handle, such as equations, tables, and graphical charts. This activity is closely related to the logical as well as the physical model of a document, since it derives constructs for one form from the other.
- C. **Viewing.** This is a device dependent mapping from the physical structure to individual output device formats. The physical model is specified in terms of high-level constructs which may need to be emulated using primitives supported by the viewing device. This usually takes the form of device drivers which translate the device independent output from the formatter into a specific printer or display format.

- D. **Recognition and Analysis.** Recognition maps from the output device domain to the physical representation, while analysis maps this back to the logical structure. These two activities are more common in the analysis than in the preparation of documents. However, this would also be needed during layout editing where alterations of the formatted image must be used to update the logical structure. For instance, deletion of a line spanning two sentences should result in the two being merged logically.

With reference to the simplified model in Figure 2-1, the work presented in this thesis emphasizes the logical model and the associated structure editing activities. Based on the above definitions, we will later introduce notions like pure formatter, integrated editor/formatter and previewer (§2.2.1).

2.1.3 Existing Document Models

We distinguish here between two classes of document architecture. We first review some document modeling formalisms which serve as a standard to which specific applications can conform. These models basically serve as an interchange standard for structured documents. The first four examples (§2.1.3.1-4) belong to this category. Then, we look at the architecture behind several specific structured systems. These latter examples put more emphasis on manipulation of structured documents than on the transmission format.

2.1.3.1 Office Document Architecture (ODA)

This standard was developed jointly by the ECMA (European Computer Manufacturers Association) and ISO (International Organization for

Standardization) for the "interchange of structured mixed-mode documents among multi-vendor office systems" [ECMA 1985] [Horak & Kronert 1984]. The standard includes definitions of an abstract structural model, a processing model and document interchange format. The ODA itself does not describe any specific system, it is a standard which applications can conform to and thus communicate.

Components in the model are specified in abstract terms. For instance, notions like *document class* are identified, but it is up to individual systems as to which classes to implement. In particular, the notion of *content portions* refers to the lowest-level logical partitions of a document (or basic objects), and are structured according to various *content architectures*. In the present version, only character and photographic content architectures have been defined for ordinary textual and pixel-based objects, respectively. Other content types, which include line graphics, spreadsheet data and voice, are intended for future extension.

Three formats are available for document interchange, namely, the *image* format which is for unintelligent hardcopy generation only, the *processable* format which conveys an unformatted and revisible logical form of a document, and the *formatted processible* format which is a formatted image with the logical structure included for revisibility. An ODA document representation is intended for machine communication with no human intervention. In view of the appearance of other final-form representations in modern printing devices (such as Postscript), the relatively simple layout model adopted by the ODA has been criticized as inadequate and obsolete [Joloboff 1986]. One application based on the ODA is the Textmaster system [Kay 1986].

2.1.3.2 Standard Generalized Markup Language (SGML)

This ISO standard defines a high-level language for describing structured documents for "interchange between different text processing systems for various purposes" [ISO 1986]. An SGML document is typically composed of two parts — markup *declaration* and markup *descriptive*. The former declares any logical constructs required (based on an SGML meta-language) while the latter describes the specific content of a document using predefined element tags or *generic identifiers*. While logical elements of a document are explicitly marked up by the originator, it is up to the recipient to decide how to process these objects. The purpose of interchanging an SGML document is therefore not confined to document preparation. The receiving end may be an intelligent document filing system, a translation system, or any application that needs to parse structured documents.

When used in publishing, an exchanged SGML source is independent of the formatters used at both ends (i.e. the author and the publisher), either of whom may have a table that maps between SGML markup constructs and appropriate codes for the formatter it uses. While page description languages like PostScript provide a device independent format between composition and printing stations, SGML serves a higher-level 'formatter independent' communication between composition systems.

SGML does not specify a particular document architecture or identify any processing activities that may take place at the originating or receiving end. Unlike ODA, however, SGML does not include a layout model or support interchange of documents as a final formatted version. Instead, a high level human-readable format is used throughout. In one application, a language has been devised for SGML conforming systems to construct conversion tables for various formatters

[LeVan & Terreni 1986].

2.1.3.3 Interscript

Developed at Xerox Palo Alto Research Center, Interscript is another formalism for representing structured documents [Ayers et al 1984] [Joloboff 1986]. The format is intended for machine rather than human manipulation. In this respect, Interscript and ODA are more suitable for applications where the revisible-form representation is not directly modified by users, but, by the system instead (as in most WYSIWYG editors). A document encoded in the Interscript base language (called a *script*) is a hierarchy of nodes which are associated with *element tags* containing attributes. However, there exist major differences from the two previous standards.

The Interscript syntax describes not only the logical structure of a document, but also constructs which may be needed in the presentation process, such as evaluation of expressions during the formatting process. An Interscript source may convey macroscopic layout controls over the formatting process, which are embedded in the document representation itself. Unlike SGML document nodes, an Interscript node may have multiple tags and thus inherit properties of various element types. Furthermore, tags may be defined to represent symbolically the layout process, in which cases attributes of a tag will serve as parameters to the layout procedures in the *pouring* process. Apart from serving as a static document representation, Interscript thus also enables specification of layout in high level terms, often called *descriptive layout*.

2.1.3.4 Document Content Architecture (DCA)

This is an IBM standard for document interchange for distributed development purposes [DeSousa 1981]. Functioning under the System Network Architecture (SNA) environment. DCA defines the format of control and information in a data stream representing a document. Three forms of DCA have been defined, namely, the *revisable-form* DCA which is editable and reformattable, the *final-form text* DCA which is a text-only formatted version, and the *final-form mixed data* DCA which contains text and images. Compared to the previously described standards, DCA puts even more emphasis on the transmission protocol than on the logical organization of a document. Working above DCA is the Document Interchange Architecture (DIA) [Schich 1982]. While DCA specifies the internal structuring of information in a document, DIA provides inter-application protocols for the manipulation of information like document distribution or document library services. It has been suggested that the roles of SNA, DIA and DCA are analogous to those of a postman, an envelope and the letter, respectively. Apart from the DCA revisable format, another IBM standard serving similar purposes is the Generalized Markup Language (GML) [Goldfarb 1981], which is in fact the predecessor of the SGML standard introduced earlier.

2.1.3.5 Kimura & Shaw's Model

One motivation of the work carried out at the University of Washington is that most systems handling multiple object types are not based on any comprehensive model which governs the structure and relations among document elements [Kimura & Shaw 1984]. The notions of *abstract* and *concrete* representation of a document are introduced. The abstract model is based on classes and instances as

commonly used in object-oriented programming. A syntax is defined for describing specific documents. The model served the implementation of a general abstract object editor [Kimura 1984].

The definition of *sequence* and *set* enables the specification of ordered and unordered object structures respectively. Another important notion is *sharing* of objects, which enables a single object instance to be multiply referenced, and is represented by a directed acyclic graph structure. *Links* are used to connect objects which are related via their data. Shared objects and links can sometimes be used for the same purpose. For instance, objects which contain identical but distinguished instances of sub-objects may be linked accordingly. As an alternative, these objects may simply share a single instance of the appropriate sub-object. Logical organization of a document, however, does not extend to the internal structuring of mathematical or graphical objects, which is the main emphasis of the work presented in this thesis.

2.1.3.6 Peels et al's Model

This model served the design of the COBATEF document system built at the Twente University [Peels et al 1985]. Three sectors of information streams have been identified as the logical structure of a document. The *primary information*, which is the essential part of a document, consists of *composite* and *basic* text elements. Examples of composite elements are chapters, sections, and subsections, while basic elements include paragraphs, lists, quotations, formulae, tables and figures. The second stream which supplements the primary information is *illustrative information*. It includes front matters like abstract and preface, back matters like appendix, and anchored objects like illustration and footnote. The last

stream is *secondary information* which facilitates retrieval of information of a document. It includes margin text, page number, table of contents, index and the like.

The logical content of a document is structured hierarchically and is specified with a BNF-like notation. There is no notion of document class, and the model applies to a universal book-style document type. Logical decomposition of purely textual elements commonly found in general texts is fairly adequate. However, figures are conceived as terminal objects and formulae are treated as a physical combination of letters, digits and special symbols.

2.1.3.7 Quint & Vatton's Model - Grif

Developed at the University of Grenoble, France, Grif is an interactive structure-oriented system whose design is based upon a formally defined document architecture [Quint et al 1986a]. The model does not distinguish between document and object classes (such as a business letter or a formula), the generic structure of both of which are specified with a declarative language called S using a Pascal-style syntax. The resulting *structure schema* enables the specific logical structure of a document or object to be created interactively. A document is modeled as a tree of *element nodes*. A node is typed and may have *attributes* which add semantics to the corresponding element. Non-hierarchical relations in the document are represented by internode *references*. The specific structure and content of a document is referred to as an *external representation*. The physical form, or the *abstract picture*, of a document is generated from that representation according to a *presentation schema*, which is specified with a language called P. For hardcopy generation, the external representation is converted to a

postprocessing formatter language source according to some conversion rules which are specified with another language F. In the present system, T_EX serves as the postprocessing formatter.

Graphical objects are not accommodated in the model. Mathematical formulae are structured and manipulated with a specialized structure editor called Edimath [Quint 1984], which works closely with Grif. However, the model for mathematical constructs is logically incomplete and physically oriented. Further discussions on this will be found in Chapter 4, when we present an architecture for mathematical expressions.

2.1.3.8 Furuta's Model

In view of the fact that a pure tree representation could be too powerful in some situations (say, for a simple block of text) but inadequate in others (such as when modeling two dimensional tables), Furuta uses a hybrid *tnt* structure [Furuta 1986]. The top level of a document tree is a strict tree structure of abstract objects, which encompass various *tree blocks* at the leaves. The internal format within these tree blocks may have a non-hierarchical free structure, which in turn may 'terminate' with either *atoms* or *transition nodes*. Atoms are actual terminating points of the entire document structure. Transition nodes lead to further strict tree portions. The name *tnt* is derived from this alternating "tree/non-tree" pattern. A meta-language resembling the BNF syntax is used to describe the architecture of strict tree portions. Only one universal structure exists for strict trees which are basically ordered lists of elements, while free structures may have any arbitrary internal format.

Manipulation of an entire document structure is supported by a group of editors. A *generalized manipulator* enables interactive, template-driven editing of strict tree portions according to the pre-defined generic structure. Other supporting editors are specialized for various free-structure tree blocks like text, table and equation. These special editors can reinvoke the generalized manipulator if strict tree portions are embedded in the corresponding tree blocks. Again, complex elementary objects are not comprehensively structured in the logical model. In particular, the interactive equation block editor is based on EQN constructs which are physically oriented [Kernighan & Cherry 1975], while structured diagrams are not accommodated.

2.2 RELEVANT CONCEPTS

We have selected four aspects in document processing for our discussion here. These concepts, apart from document modeling itself which has just been discussed, are considered to be relevant to our work.

2.2.1 The Edit-Format Cycle

One useful criterion in classifying document systems is based on the coherence between the editing, formatting and viewing activities as described in Section 2.1.2. In this respect, two classes of systems exist at the extremes.

- A. Batch formatters (also known as pure formatters or document compilers) accept a document specification which is based on a source language, and generate a physical output format accordingly. The specification carries the

intended document content intermingled with formatting (or markup) commands, and is prepared with a separate ordinary text editor. Referring to the standard model in Figure 2-1, the editing, formatting and viewing processes would be disjoint and non-concurrent activities in a batch formatter. Prominent examples include T_EX [Knuth 1984], Scribe [Reid 1980b] and the Troff formatter [Kernighan & Lesk 1982].

- B. Interactive editors (also known as integrated editor/formatters, or WYSIWYG editors) provide a closed edit-format-view cycle. Instead of specifying through a language, a user interacts directly with the target document image on a high resolution screen, which simulates the final printed document in terms of typefaces, line weights, and geometric proportions among objects. This philosophy is commonly treated as direct manipulation, in contrast to the source-language model described above. Typical examples include Etude [Ilson 1980], Xerox's Star [Smith et al 1982b], Lara [Gutknecht 1985], Interleaf [Morris 1985] and countless commercial systems.

Both approaches have their own merits, a discussion on which will be found in Section 2.3.3.1 where we will also look at recent research on providing a compromise philosophy. Although the two approaches described above are most common, language-based formatters do exist which work incrementally to provide screen preview which cannot be modified anyway [Chamberlin et al 1982]. In other words, WYSIWYG editors do not necessarily imply direct manipulation (see also §2.3.3.1), the latter being a more general concept in user interface modeling.

2.2.2 Content-Style Separation

We now look at another criterion used in rating document systems. Here, the question is how well a system separates the specification of the logical content of a document from that of the style for presentation. Again, we first describe two classes at the extremes. A recent comprehensive article on this issue can be found in [Johnson & Beach 1988].

- A. **Procedural systems** require a user to specify procedurally all operations needed to achieve all final effects in a document. One has to issue precise commands at appropriate points. For instance, the commands for a page break, bigger point size and centering must accompany all chapter headings. One is therefore responsible for ensuring a consistent style throughout the document. Furthermore, the logical structure of a document is not recorded since logical elements are represented by some low-level typographical commands.
- B. **Declarative systems** enable a user to identify high-level document constructs with appropriate tags instead of instructing the system about specific actions. The user thus specifies the *content*, but not the *style*, of a document. This content-format separation not only ensures uniformity of style, but also introduces flexibility. The same document specification can yield outputs in a totally different style with minimal efforts. The power of this is the ability to alter the representation of a particular tag throughout a document, by modifying the style associated with the tag.

Two early examples of batch declarative systems are the GML [Goldfarb 1981] and Scribe [Reid 1980b] formatters, while interactive counterparts include the Tioga [Teitelman 1983], Star [Smith et al 1982b] and Etude [Hammer et al 1981] editors.

Commercial WYSIWYG systems which provide style mechanisms did not appear until recent years. Examples include Interleaf, Xerox's Ventura Publisher, Microsoft's Word and Frame Maker [Johnson & Beach 1988].

Purely procedural formatters do not exist in practice; the separation of content and format has been achieved in otherwise procedural formatters (like Troff on UNIX) via the use of macros. The philosophy is however different from a truly declarative system. The declarative rationale demands no typographic expertise from the user, that being provided by the system, or more precisely some experienced personnel. In the procedural approach, a user has to write the required macros and thus has to judge style or aesthetic issues. Though macro packages are often provided, these are not as comprehensive as, say, the style database in Scribe (see §2.3.1.2). One exception perhaps is the L^AT_EX macro library for T_EX, which is based strongly on Scribe notations [Lamport 1986].

There are three basic notions in a declarative system. A *document class* refers to a category of documents sharing the same style or actions for presentation of document elements. Each class is associated with a set of *abstract objects*, each of which possesses a *presentation style* which specifies detailed typographic actions needed for its presentation. Using the terminology in [Johnson & Beach 1988], these three correspond to style sheet, style rule name and style rule effects, respectively. The document class *thesis*, for instance, may include the object *chapter heading* whose style may imply centering, turning bold, and raised point size. Table 2-1 provides a cross reference for notations used in various declarative systems.

Although a declarative formatter has knowledge of a document's logical structure, structure editing is not implied. For instance, suppose a user wants to delete a *theorem* environment while editing a Scribe source file with a simple text

SYSTEM	NOTATIONS		
	Document class	Abstract object	Presentation style
Scribe	document type	environment	state parameters
Andra	style	style element	attributes
Janus	document profile	tag	tag routines
Tioga	style	look & format	format definitions
Grif	document class	object class	presentation schema
ODA	document class	logical object type	layout directives
Ventura	style sheet	tag	tag's attributes
Star	see §2.3.1.3	property sheet	property values

TABLE 2-1. Cross references for notations in declarative systems

editor, he still needs to locate explicitly its begin and end tags before removing the text in between. Structure editing, which will be introduced next, would allow a one-step selection of the theorem as an individual logical object.

2.2.3 Structure Editing

We now describe another form of editing which also requires the system to know a document's logical structure. Structure editing enables a user to manipulate logical rather than primitive units in a document, that is, the structure as well as content can be organized conveniently. Structure editors are sometimes called *hierarchy editors*, since most of these organize a document as a tree of elements. The following are typical features of a structure editor.

- A. **Logical selection.** A user can select logical elements naturally. For instance, to select a section for deletion or copying, one need not struggle through the text in order to locate its boundaries. Instead, a section would exist as a single *addressable* unit. This enables convenient manipulation of the

structure of a document.

- B. **Logical navigation.** One can move around the text in logical steps, such as moving from a selected section to the first, the previous or the last section at the same level, to a particular subsection under it, or to the chapter to which it belongs. This enables convenient navigation through large documents.
- C. **Viewing options.** Since a document is arranged as a hierarchy of objects, the user can specify any level of abstraction when viewing a document. Details below the specified level would be hidden. This global or outline view is very useful in managing large complex documents, and is available in modern word processors with outlining features.
- D. **Template-driven editing.** The system typically provides templates for logical document elements, based on its knowledge of their composition and layout style. The use of templates eliminates a user's need to enter information already implied in the document model, resulting in a declarative specification. Furthermore, syntax direction is normally provided and the structure of objects can be validated upon their entry.

Structure editing appeared initially in program development systems, where a document's logical model (i.e. syntax of the language in question) was well defined. Examples of programming environments include the Cornell program synthesizer [Teitelbaum & Reps 1981], the Mentor [Donzeau-Gouge et al 1984] and the Gandalf [Medina-Mora 1982] project. Hierarchy editors for general documents include Walker's Document Editor [Walker 1981], PEN [Allen 1981], ED3 [Stroemfors & Jonesjoe 1981], SDS [Fraser 1981], and XS-1 [Beretta et al 1982]. Structure editors which also qualify as hypertext systems are reserved for the next section.

Apart from pure textual contents, structure editing has also been applied to graphically oriented objects like mathematical formulae [Quint 1984], tables [Biggerstaff et al 1984] and structured diagrams [Jones 1986]. Structure editors are typically tuned to a particular class of objects. LEXX is an example which aims at the programmability for a variety of structured but basically textual data [Cowlshaw 1987].

2.2.4 Non-linear Text Models

Here, we briefly introduce a class of systems which model documents in a non-linear fashion. Better known as hypertext systems, these emphasize dynamic on-line presentation of a document rather than using a linear static media like paper. The inspiration of these systems can be traced to a paper titled 'As we may think' written by Vannevar Bush [Bush 1945]. Bush envisioned a non-computer-based system called *memex* in which "individuals would store all their books, records and communications, establish trails through them, and rapidly retrieve specific information contained in them".

Two of the earliest systems influenced by Bush's ideas are the NLS (oN-Line System) developed at the Stanford Research Institute [Engelbart et al 1968] and the HES (Hypertext Editing System) developed at Brown University [Carmody et al 1969]. The term 'hypertext' was invented by Ted Nelson (a co-author of HES) to address computerized memex-like systems. It broadly refers to any non-linear text being arranged as a network of nodes connected by directed and typed links (or paths). Each node holds an arbitrary amount of textual or graphical content. A user browses through a hypertext following existing links and modifies the node content and link structure accordingly.

Both NLS and HES have evolved into commercial products, namely, Augment [Engelbart 1984] and Xanadu. Existing applications of hypertext cover on-line paper critiquing and refereeing such as TextNet [Trigg & Weiser 1986], idea processing and collaborative work such as Xerox's NoteCards [Halasz et al 1987], electronic encyclopedia as in [Weyer & Borning 1985], and database for CAD applications such as Neptune [Delisle & Schwartz 1986]. An introductory but comprehensive survey on hypertext systems can be found in [Conklin 1987].

More recent interests in hypertext applications have centered around the concept of computer-supported cooperative work (CSCW). The objectives are to provide computer support for group activities like collaboration, communication, brainstorming, and cooperation. Despite Douglas Engelbart's earlier work on providing collaboration support for 'knowledge workers' [Engelbart et al 1968], the first conference on this topic did not take place until recently, as technology for hypertext systems matures [Bair 1987]. Due to their nature, these collaboration tools with multi-user interfaces are sometimes quoted as "what you see is what I see" (WYSIWIS).

Multimedia documents are basically hypertext in which node contents are extended to encompass elements in other media (such as digitized sound and video animation). These are alternatively known as *hypermedia* systems. Previous work includes Agora [Naffah & Karmouch 1986], CCWS [Poggio et al 1985], Concordia [Walker 1988], GENIE-M [Angell et al 1986], Intermedia [Yankelovich et al 1988], and MINOS [Christodoulakis et al 1986]. Commercial hypermedia products have started to appear recently; a notable example is the HyperCard which basically serves the development of other hypermedia applications on the MacIntosh [Goodman 1987]. There have been many suggested and demonstrated scenarios exploiting these hyper or dynamic documents, in which the viewing flexibility can

never be achieved with paper documents. Despite this, it would still be a long way before these documents could entirely replace the paper media (if that will ever happen).

2.3 EXISTING SYSTEMS

Here, we first review three systems which have had definite impact on the development of today's document system. Next, we will examine the trends of current commercial systems. Finally, we look at the current interests of researchers in document processing. For further information on systems review, the reader is referred to two comprehensive survey papers on document preparation systems [Furuta et al 1982] [Meyrowitz & van Dam 1982].

2.3.1 Influential Systems

We now review three systems which have influenced later developments. These all appeared around the 'batch to interactive' transition (see §1.1). T_EX and Scribe, two of the last batch formatters developed, have promoted high quality computer typesetting and declarative document specification, respectively, while Xerox's Star has triggered the WYSIWYG trend.

2.3.1.1 T_EX

Developed at Stanford by Donald Knuth [Knuth 1979], T_EX has special emphasis on so-called aesthetic aspects both in formatting general text and setting equations (which is a major integral part of the system). One of the objectives of T_EX is that

computer typesetting should improve, not degrade, the quality achieved in traditional typesetting. The success of T_EX in handling complicated equations can be reflected by the American Mathematical Society's adopting it as their standard publishing system. Furthermore, interactive systems which pursue the finest layout quality often employ T_EX as a postprocessing formatter.

The creation of T_EX is accompanied by important contributions both to typesetting and computer science in general. These include:

- Metafont, a constraint-based graphics system for font design [Knuth 1979],
- a box-and-glue physical page model featuring stretchable glues [Knuth 1984],
- an algorithmic approach to optimize line breaking [Knuth & Plass 1980] and pagination [Plass 1981], both employing look-ahead and dynamic programming techniques, and
- the WEB system for literate programming (that is, an integration of programming and documentation); a WEB file contains intermixed T_EX and Pascal commands [Knuth 1983].

2.3.1.2 Scribe

Developed at Carnegie Mellon by Brian Reid [Reid 1980b], Scribe is now a commercial product of Unilogic Ltd. [Unilogic 1983]. Scribe was the one of the first truly declarative formatters. Most importantly, the style mechanism was not achieved with user managed macros, but with the system's built-in knowledge [Reid 1980a].

A Scribe user specifies the content and structure of a document, without any necessary concern about design and layout issues. Scribe stores the latter knowledge in a style database, which includes definitions of various *environments* and their associated typographic effects. This database is normally maintained by printing experts. A user declares the structure of a document using the available environments, while the system consults the database for a consistent and professional layout style.

Another feature of Scribe is a writer's work bench, which is implemented as an integral part of the formatter. This provides facilities like symbolic bibliography citation and cross referencing, and the automatic generation of index, bibliography, table of contents and list of figures. These are typically achieved in other formatting systems through the use of macros and separate pre-processors.

Scribe and T_EX aim at two different groups of users. While T_EX provides complete low-level control in the formatting process, Scribe emphasizes high-level structuring of a document and hides detailed control from the user. The discrepancies between declarative and procedural systems (§2.2.2) implies a possible trade-off between abstraction and control. L^AT_EX is a macros package for T_EX which simulates Scribe's high-level environments while achieving T_EX's flexibility and user control (and typesetting quality) [Lamport 1986]. Furthermore, another interactive system, Etude, is greatly influenced by Scribe's notions [Hammer et al 1981].

2.3.1.3 Star 8010

At about the same time when T_EX and Scribe were developed, one of the first commercial interactive systems appeared. An advanced office automation product

developed at Xerox Palo Alto Research Center [Smith et al 1982b], Star was a successor to an earlier experimental system Bravo implemented around 1975, from which most of the innovative ideas were inherited. Star simulates the image of a final document on a high-resolution screen. Apart from the WYSIWYG emphasis, other important inspirations in user interface design were embodied in the product. For instance, the desktop or physical office metaphor, drop-down menus, and the associated direct manipulation techniques are found in most interactive systems today. MacIntosh and GEM applications follow startlingly similar principles.

Objects in a Star document contains objects belonging to various content types, such as character, paragraph, illustration or page. Each object is associated with an appropriate set of properties (depending on its type) which govern its presentation style. Properties of an object are modified via a property sheet which corresponds to a single style rule (and is therefore different from a style sheet which usually refers to a *collection* of style rules [Johnson & Beach 1988]). However, since property values are associated with each object 'instance' rather than an object 'type', Star is not considered as a truly declarative system. For instance, each paragraph has its own set of property values, instead of all paragraphs sharing a single property sheet. Systematic or global changes to the presentation style are therefore difficult. Despite this fact, Star has many influential features as an interactive document system.

2.3.2 Commercial Developments

Instead of giving a full account of individual products, we will only focus on the trends of current systems. A fairly 'recent' development of considerable commercial importance is *desktop publishing* (DTP). The term was invented by

Paul Brainerd of the Aldus Corporation, and the 'phenomenon' did not really take place until the introduction of the LaserWriter by the Apple Corporation in 1985. It refers to the use of a personal computer for interactive composition of documents incorporating graphics and images, from which the output can be produced economically on a local printer or alternatively on a remote typesetter. A more precise description of the activity can be found in [Seybold 1987]. Countless DTP applications have appeared over recent years. Unlike the traditional electronic publishing industry, in which fully integrated systems are most common, desktop publishing applications fall into several categories according to their functions.

- A. **Main composition packages.** These are responsible for basic text formatting and page composition, providing most necessary typographic controls which are typical in the typesetting industry. These normally only provide simple word processing or diagram preparation facilities, but usually allow import of these objects prepared with other packages. Integration of text and graphical materials is often achieved through this process. Examples are Aldus' PageMaker, Xerox's Ventura Publisher, Apple's MacPublisher and Adobe's Illustrator.
- B. **Word processing packages.** These supplement page composition packages with powerful text manipulation facilities. Typical word processors are Microsoft's Word, WordPerfect, WordStar, Multimate, MacWrite and many others.
- C. **Graphics packages.** These include general drawing packages like MacDraw, MacPaint, GemDraw and image acquisition programs.
- D. **Specialized packages.** Most special-purpose applications also produce output formats which can be imported to composition packages. Examples

include database, spreadsheet, presentation graphics and CAD applications.

Although desktop publishing softwares are dominated by WYSIWYG applications, batch systems do exist. For instance, there are various microcomputer versions for the $\text{T}_{\text{E}}\text{X}$ formatter; these would suit documents that contain little graphics but many equations. The trend of splitting tasks among independent but compatible packages has resulted in a high degree of flexibility — a definite advantage over the integrated, structured approach (which will be discussed in §2.3.3.2).

2.3.3 Current Research Emphases

The current interests to be examined here are, first, merging the power of interactive and batch systems, and second, document structuring and object integration in interactive systems. Other current activities which will not be discussed here include computer-supported cooperative work (CSCW), hypermedia systems, multilingual document processing, and document interchange standards.

2.3.3.1 Batch-or-WYSIWYG Dilemma

Although interactive systems have existed for more than a decade, there is a reluctance to abandon batch systems totally. Let us first look at the factors one has to consider when deciding on the approach, and then discuss various solutions to the problem. The situation is best illustrated with arguments put forward by advocates of both approaches, which can be summarized as follows.

A. Pro-WYSIWYG (or Anti-Batch)

1. **Interaction Model.** The model of interaction in WYSIWYG systems is based on direct manipulation of the target document. A language-based approach forces a user to describe two-dimensional complicated objects in a linear form.
2. **Production Cost.** Isolation of the edit, format and view processes in a batch system implies a long edit-compile-debug cycle. Waste of time and resource is inevitable when complex objects (like equations or diagrams) are involved, due to the lack of document preview.
3. **Incremental Formatting.** Text entered or modified in a WYSIWYG editor is incrementally reformatted, where usually only the immediate paragraph is involved. Batch systems normally have to reformat an entire file even after minor changes.

B. Pro-Batch (or Anti-WYSIWYG)

1. **System Requirement.** A strength of batch formatters is that any ordinary terminal can be used. This, as opposed to a WYSIWYG editor's need for at least some sort of 'workstation', has provided flexibility and economy.
2. **User Control.** A batch system user 'programs' and controls the formatting process via an expressive language. The functionality of a WYSIWYG editor is shielded from the user since the friendly interface permits only a limited set of operators.
3. **Quality.** Without the need for 'instant' feedback, batch systems can afford to use formatting algorithms optimized for quality. To improve print quality, some WYSIWYG editors provide off-line formatting

optimizations (as in Tioga), while most others rely on a postprocessing formatter (as will be discussed later).

In fact, there used to be another criticism that interactive systems, while maintaining only the concrete form of a document, totally disregard its logical structure. However, recent WYSIWYG editors have demonstrated the separation of content and structure by incorporating style mechanisms.

One of the earliest inspirers of WYSIWYGery, Douglas Engelbart, envisioned a system having "... display terminals with a capability for the ranges of fonts available on the photocomposer ..." well before the first such system was built [Engelbart et al 1973]. In a recent paper on his editor Augment, however, Engelbart addresses the deficiencies of these systems as the 'WYSIWYG syndrome' [Engelbart 1984]. It is also tempting here to quote Kernighan's famous interpretation of WYSIWYGery as "what you see is all you've got" [Furuta 1986]. Some researchers have worked towards providing some form of co-existence of the two philosophies. There are basically two approaches to achieving this.

2.3.3.1.1 A Weak Approach

- A. **Batch dominant.** Here, a page previewer displays on a bitmapped screen the output from a batch formatter, and thus simulates the WYSIWYG effects [Chamberlin 1987] [Chen et al 1986b] [Trevorrow 1986] [Chamberlin et al 1982]. Modification can however only be made to the unformatted source file. This is an example of a WYSIWYG situation yet without direct manipulation.
- B. **WYSIWYG dominant.** Here, a truly WYSIWYG editor compiles its output in an existing batch language, which is then reformatted by the appropriate

system [Cowan & Smit 1986] [Crisanti et al 1986] [Furuta 1986] [Quint et al 1986b]. The user thus gets a WYSIWYG feel on the screen and batch quality on the paper. The screen version, however, may have different line or page breaks.

2.3.3.1.2 An Ambitious Approach

This approach aims at an equal share between the batch and interactive characteristics of a system. A prominent project with this goal is 'Visually ORiented T_EX' (or VOR_T_EX). Developed at the University of California, Berkeley, it also serves as an enhanced environment for T_EX using SUN workstations. The first phase of the project, which has been completed, includes a screen previewer for T_EX's standard output (i.e. batch-dominant approach) [Chen et al 1986b].

The second phase, which is still in progress at the time of this writing, aims at a hybrid 'batch & WYSIWYG' model [Chen et al 1986a]. A document would have a *dual representation*, namely, the unformatted source description and the formatted WYSIWYG image. Both representations could be edited on separate windows; when one is modified the other would be updated automatically. In a separate article, the authors analyze nine different tasks in a document development environment and, for each of these activities, present The relative merits of the source-language and direct-manipulation approaches [Chen & Harrison 1988].

2.3.3.2 Structured and Integrated Systems

While commercial composition systems often support and rely on the import of externally prepared document objects, some researchers in the field are taking quite a different approach. In these projects, a structural view is imposed on documents

containing mixed object types, and emphases are being put on integrating the processes which produce these elements. The goal is towards some form of integrated, structured and interactive system. A typical approach is based on the definition of an appropriate document model, which serves as a unified framework for various document elements and activities, that is, an architectural design approach. Generally speaking, this can be considered as an extension of the structured editing concept to a more diverse class of complex, non-textual objects. Notable projects include Furuta's work [Furuta 1986], Grif [Quint et al 1986a], Tioga [Teitelman 1983], Andra [Gutknecht & Winiger 1984], and Kimura et al's work [Kimura & Shaw 1984].

Most work on multimedia systems are closely related to this concept, especially with the notion of integrated media editors and virtual editing in multimedia documents [Poggio et al 1985] [Maxemchuch & Wilder 1981]. However, like the previously mentioned projects, we have here aimed at defining an architecture for more traditional documents (i.e. dealing only with the static medium).

Although the work presented in this thesis shares similar objectives as other document modeling formalisms, it aims at a more complete logical decomposition of elementary objects (which we feel is lacking in existing models). The three objects which will be studied in detail in this thesis are equation, table and structured diagram.

2.4 CONCLUSIONS

In this chapter, we have presented the fundamental concepts in document processing, including document modeling which is the main theme of this thesis.

We have also reviewed some prominent systems and research activities. In the next chapter, we will present the overall design of a document architecture.

Chapter 3

Overall Document Architecture

In this chapter a design is presented for a comprehensive logical document model. As the idea of document architecture is not new, we will discuss in detail only those aspects which are emphasized in our work here; others will be given a brief discussion with appropriate references. We will first give an overview of the model, which is then presented as three major components, namely, the logical, physical and processing models. Finally, we will discuss an integrated and structured system whose design could be derived from the model.

3.1 INTRODUCTION

We first discuss the scope and emphases of our document model. Then follows an overall picture of the architecture. Finally, we introduce the notation used later for its formal presentation.

3.1.1 Scope and Emphases

The purpose of defining a document model here is to provide a basis for the design of an integrated and structure-oriented document system. Unlike architectures which mainly serve as a standard, such as the ODA or the SGML, no attention is paid here to the transmission format of document elements. Furthermore, we only introduce general notions, without any reference to a particular implementation. Consequently, only the architecture of 'universal' elements, which are non-specific

to any particular document type, will be studied in detail.

We are basically interested in the logical aspects of the model, while little emphasis is put on layout issues. This is justifiable since graphics-oriented page models are well developed [Warnock & Wyatt 1982] and, as will be discussed below, it is the logical organization of existing document architectures that needs further attention. The model presented here therefore does not include all formatting constructs which would be needed for a complete implementation.

Particular attention is paid to logically oriented materials such as structured scientific documents containing equations, tabulated data, technical diagrams, and the like. On the other hand, layout-oriented materials like advertisements or posters, which do not have an inherent logical structure, will not benefit much from the model or, in fact, from structured editing itself. Furthermore, document objects in dynamic media, such as voice annotations and animated illustrations, are not accommodated in our model.

As we have reviewed in Section 2.1.3, there already exist a number of document architectures. In defining our model here, special emphases have been put on the following aspects.

- A. **Complete Structuralization.** Existing models have overlooked the internal structuring of lower-level objects in a document hierarchy. There has been a tendency to concentrate on high-level organization of a document, without specifying adequately the logical architecture of objects like equations, tables and figures (see §2.1.3). Consequently, while most existing systems support structural manipulation at a global level (such as automatic renumbering of sections, or generation of tables of content), complex objects are not handled with the same degree of flexibility. Here, the logical organization of a

document proceeds down into these objects. This continuity in the model would enable a unified rationale of *structured editing* at all levels. The detailed architecture of three fundamental objects, namely, *equation*, *table* and *structured diagram*, will be presented in the next three chapters.

- B. **Process Integration.** Apart from integration of objects appearing in a document, we are also concerned about integration of the *processes* that produce these objects. In a batch system, objects are typically handled by dedicated preprocessors, which often result in constraints in nesting among various objects, as in UNIX [Kernighan 1984]. In an interactive system, objects are often prepared with separate packages and then imported to a main composition process for integration. This has undoubtedly resulted in high flexibility, but still it would be desirable to have an integrated and structured process for handling various object types. The unified model defined here has aimed at enabling editors specialized for various objects to work in an integrated and coherent manner.

3.1.2 An Overview

One may find Figure 3-1 useful for our discussion here. At the highest level, a document is viewed as a hierarchy of abstract elements here called *composite objects* (such as the front matters and the main body of a book). These objects are further decomposed into other composite elements, and eventually reduce to the basic units in a document, such as equations and tables, which are here called *elementary objects*. The internal organization of these elementary objects are not shown in the figure. This entire hierarchy of composite and elementary objects, together with some *non-hierarchical* relations among them, constitute the

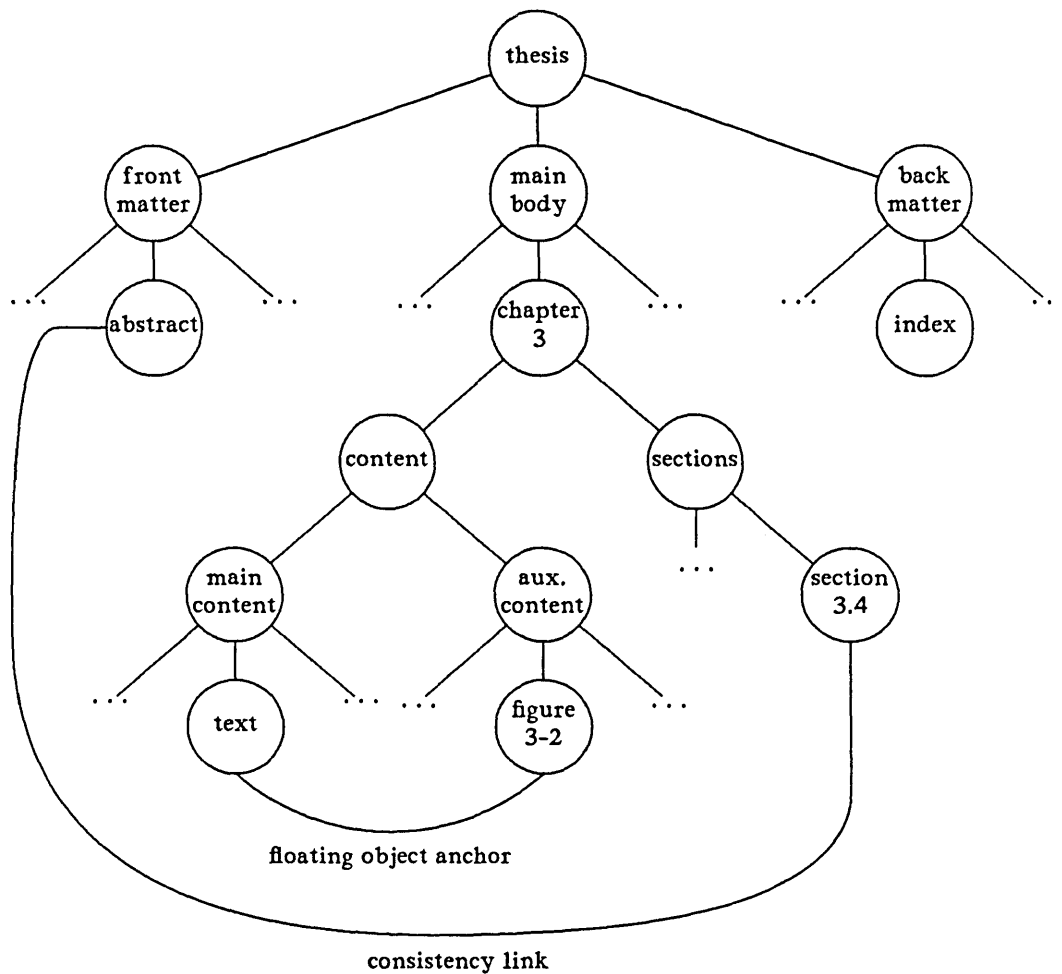


Figure 3-1. The global structure of a document

fundamental structure of a document. These will be considered in Section 3.2.1.

The fundamental component represents the basic content and structure of a document and, as will be seen later, would serve both editing and formatting activities. Superimposed upon this structure is a set of auxiliary constructs which are basically non-hierarchical relations inspired by hypertext models. These

constitute the *auxiliary structure*, and will be described in Section 3.2.2. These constructs basically serve the development as well as dynamic presentation of a document, One example of this structure shown in the figure is the consistency link, which will be described later.

3.1.3 Notation

A formal presentation of the overall document model can be found in Appendix A, which will be useful in later discussions. The following notation, which is based on the Backus-Naur Form (BNF), will be used in the appendices as well as throughout the chapters.

$\langle . . \rangle$	an object type identifier
$\langle . . \rangle ::= \dots$	composition of an object type
$\dots , \dots$	an ordered sequence
$\dots + \dots$	an unordered set
$\{ \dots \}$	a self-repeating sequence
$[\dots]$	a self-repeating set
$[\dots]$	an optional component
$\dots \mid \dots$	a selection
$\dots \rightarrow \dots$	a non-hierarchical link

Two demonstrative usages of this notation in the context of defining a document and a mathematical construct, respectively, are as follows :

`<chapter> ::= <title> , {<composite block> | <text block>} , {<section>}`
`<summation> ::= <summand> + <index> + [<lower bound> + <upper bound>]`

3.2 LOGICAL MODEL

As we have summarized before, the logical model is partitioned into a fundamental and an auxiliary component. After discussing these two structures, we will talk about some rules which govern creation of their instances. One may refer to the formal description of the overall document model in Appendix A for the following discussions.

3.2.1 Fundamental Structure

As mentioned earlier, the fundamental component of the logical model embodies the basic logical content of a document and, unlike its auxiliary counterpart, contains information which is required in the formatting process. As depicted in Figure 3-1, a document is here perceived as a hierarchy of elements among which non-hierarchical relations are established. Referring to Appendix A, the hierarchical composition at each level is based on either an ordered sequence or an unordered set of subordinate elements. Corresponding to the nodes in a document hierarchy are objects which fall into two types. Composite objects are high-level and abstract entities while elementary objects correspond to basic units appearing in a document. Let us now study these two objects in greater details.

3.2.1.1 Composite Objects

Composite objects are here defined as abstract collections of elementary and/or composite objects. The entire document itself is a composite object, though quite a special one. The first definition in the model thus specifies the various document types the model supports (see Appendix A). The object *document* is then defined in terms of other composite objects, such as theorems, quotations, enumerated lists, and chapters. Composite objects correspond to distinct nodes in a document tree (i.e. an instance which conforms to the model), and would be addressable for manipulation during structure editing.

The definition of a composite object in the model specifies its allowed composition, as illustrated in Appendix A. In Section 3.4.2, we will introduce formatting rules which would specify any typographical actions associated with each composite object, and the way in which its various components would be combined. A composite object definition is similar to a *style rule* in declarative systems, using the terminology in [Johnson & Beach 1988]. However, apart from the associated style rule effects, the former also specifies the logical composition (i.e. architecture) of the object in question. This would be needed in template-based document editing.

A document template would be similar to other templates for ordinary composite objects. However, a document type is also associated with its own definitions of other composite objects belonging to documents in that category. The same composite object may thus be defined in different ways for various document types, in terms of presentation styles and logical composition. The former flexibility is provided via style sheets in declarative systems.

As mentioned before, the scope of our model covers only those structures which are not specific to any particular document type. We will focus more on objects which are universal to different documents, that is, elementary objects. For illustration purpose, we decompose the document type *book* and its related composite objects in Appendix A. The definitions, however, terminate at the elementary object level. The internal architectures of elementary objects, which will be defined below, are instead listed separately in Appendix B onwards.

3.2.1.2 Elementary Objects

Elementary objects are here defined as the fundamental units in a document, such as equations, tables, text blocks, line graphics, and scanned images. Residing at the bottom levels of a document hierarchy, these objects have some unique logical and/or physical properties, which implies specialized editing and formatting facilities. This is the basic distinction between these objects and their composite counterparts described previously. Two classes of elementary objects, structured and unstructured, are defined here.

3.2.1.2.1 Structured Elementary Objects

These are logically oriented objects which have an inherent architecture. Structure manipulation and syntax direction could therefore be provided accordingly. Furthermore, since there exists a well defined relationship between their logical and physical forms, the 'edit-format' philosophy can be applied to these objects. A user would therefore manipulate the logical structure, while the physical form would be generated automatically. The detailed model for three of these objects, as indicated below, will be presented in the following chapters. We feel that those three deserve further investigation.

- A. **Text.** This refers to ordinary text which is simply organized as a hierarchy of paragraphs, sentences, words and characters. Natural language structure is not accommodated in the model.
- B. **Program.** This refers to program source text, and is modeled according to the syntax definition of the language in question. Specialized language-sensitive program editors first appeared in programming environments.
- C. **Equation.** This refers to mathematical expressions often found in scientific documents. The architecture and a structured editor implemented for this object will be presented in Chapter 4. The architecture definitions are listed in Appendix B.
- D. **Table.** This refers to tabulated structures which may contain other elementary objects of arbitrary type. A model for this object, which is based on the relational structure, will be presented in Chapter 5. The logical decomposition can be found in Appendix D.
- E. **Structured diagram.** This class of diagrams have an inherent structure, both logically and geometrically. The logical significance of these diagrams distinguishes them from their unstructured counterparts described below (which may well be structured physically). A study on achieving a generalized model for this object will be presented in Chapter 6.

As will be seen in Chapter 5 and 6, both tabular objects and structured diagrams can be data-driven, in which case their content might be associated with an underlying database. This would enable dual representation of data in tabular and graphical forms, which is commonly available in existing packages.

3.2.1.2.2 Unstructured Elementary Objects

These elementary objects do not have an inherently defined logical structure. Consequently, the 'edit-format' philosophy would not be appropriate for these objects. Instead, an editor would handle directly the physical form of an object — the only form it can assume. However, layout-oriented editors for these objects could still provide specialized functions. The following unstructured objects have been defined.

- A. **Image.** Sampled images are simply modeled as a two-dimensional array of pixels. A specialized editor for this object would probably provide typical image processing functions.
- B. **Free-form diagram.** These diagrams do not have an underlying logical structure. Despite the lack of diagram syntax and semantics, these objects may however be structured geometrically, and therefore benefit from object-oriented drawing tools such as MacDraw.

3.2.1.3 Non-hierarchical Links

We have studied the objects composing the fundamental structure of a document. The relations among these objects described so far are basically hierarchical. We now talk about non-hierarchical links superimposed on the tree structure which convey other logical information about a document. It should be noted that the constructs described here still belong to the 'fundamental' component of the document model. Their formal definitions are again listed in Appendix A.

- A. **Cross reference link.** This connects a referencing text point to its quoted object, which may be composite or elementary. The location of the latter

would be inserted at the former during formatting. An example would be the quotation of a tabular object from within a text block.

- B. **Content sharing link.** This connects two objects, which must be of the same type, whose contents are intended to be always identical. This would serve duplication of identical objects such as universal formulae. More than two objects could be linked accordingly for this purpose.
- C. **Floating object anchor.** This connects a referencing text point to a floating object, and would be used to position the latter with respect to the anchor point according to some constraints (such as maximum pages of separation). The appropriate reference tag (such as a figure number) would be generated.
- D. **Footnote link.** This connects a quoting text point to a footnote object, and would be used to position the latter on the same page as the quoting text. The appropriate reference tag would be generated.
- E. **Index link.** This connects a text point to an index entry, a composite object residing within the index section, and would be used in automatic index generation.
- F. **Bibliography link.** This connects a text point to a bibliography entry, a composite object (residing in the bibliography pool) which holds information for a reference. The appropriate reference would be generated.

3.2.2 Auxiliary Structure

We have presented the fundamental components of the model which conveys the basic logical content of a document. The auxiliary structure we are going to

discuss is intended to provide flexibility found in typical hypertext systems. Though residing alongside the fundamental structure in the document tree, this component would be ignored during the formatting process (i.e. for hardcopy generation), but instead serve only the document development and dynamic viewing process. Refer to Appendix A for the definitions of the constructs described below.

3.2.2.1 Writing Support

Two structures have been included to provide assistance to the author when developing a document.

- A. **Idea pool.** This is a composite object with arbitrary structure and content. An idea pool could be attached to any node in the document tree, and would contain ideas to be incorporated *somewhere* within the corresponding object (to which it is attached). Ideas could be moved to lower-level document objects, or even become part of the fundamental structure itself, as they become solid. For instance, sketchy comments could initially be attached to a chapter node, but eventually moved to the appropriate subsection or even paragraph. This structure would thus support brainstorming as well as top-down writing strategies.
- B. **Consistency link.** This connects two objects of arbitrary type, composite or elementary, which must be consistent in their implication. It serves to remind the writer to make other necessary changes when any of these objects is modified. For instance, a table and all subsections which comment about its data could be linked accordingly.

3.2.2.2 Reading Support

We now discuss some constructs provided to present a document as a dynamic book rather than as a linear paginated view. Four links defined previously in the fundamental structure would also serve this purpose. These are namely the bibliography, index, cross reference and floating object links (see §3.2.1.3).

- A. **Notepad.** This is structurally similar to the idea pool described above, though the two would be separately maintained by the writer and the reader. The user could attach the pad to any object to which the notes refer.
- B. **Bookmark.** This flag could be attached to a composite or an elementary object, or even within an elementary object. The latter would be useful for, say, marking a statement in a program or an entry within a table.
- C. **Relevancy link.** This would connect relevant objects, such as when a section is pre-requisite to another. Unlike other reading facilities, which are maintained by the reader, this would normally be constructed by the writer accordingly.
- D. **Reading path.** This links together a sequence of objects that have been visited by the reader. Since the document is no longer read in a linear order, this would help the reader to view globally those objects not visited yet, or to revisit previous objects.

3.2.3 Instantiation Rules

We have already described the logical composition of a document, including both the fundamental and auxiliary structures. The global structure of an example

document, excluding the internal organization of elementary objects, is shown in the previously discussed Figure 3-1. We now introduce two set of rules which might be used to discipline the creation of a specific document. These basically provide possible syntax direction during manipulation of the global structure of a document. We only focus on nesting elementary objects and link creation. On the other hand, the scope of composite objects is implementation dependent, and is not discussed here.

OBJECT TYPE	CAN NEST				
	table	diagram	equation	program	text
table	yes	yes	yes	yes	yes
diagram	yes	yes	yes	yes	yes
equation	no	no	yes	no	yes
program	no	no	no	yes	yes
text	no	no	yes	no	yes

TABLE 3-1. Nesting rules for elementary objects

3.2.3.1 Elementary Object Nesting

These rules govern the creation of one elementary object within another, and determine which specialized editor could be invoked from each other. These are meant to provide convenience, and only combinations which seldom occur are excluded in Table 3-1. There should be no technical limitations regarding inter-object nesting since, as will be seen later, an embedded object would simply be a black box to its parent.

3.2.3.2 Inter-object Links

This set of rules specify the types of objects which can be connected via the various links described in Section 3.2.1.3 and 3.2.2. Table 3-2 lists the source and destination object types for various links. Syntax direction could be provided accordingly during global structure manipulation of a document.

LINK	OBJECT TYPE	
	source	destination
bibliography	text	bibliography object
cross reference	text	any
floating object anchor	text	floating object
content sharing	any	same as source
footnote	text	footnote object
consistency	any	any
relevancy	any	any
index	text	index object

TABLE 3-2. Connection rules for inter-object links

3.3 PHYSICAL MODEL

We have introduced the logical components of the document model. The physical model is defined here with reference to its logical counterpart. Furthermore, due to our emphasis on logical organization, the simplified model presented here will not suffice for implementation of a complete formatting system.

Page-oriented documents are organized into non-overlapping page units. A page is further organized into a hierarchy of composite areas, which eventually

terminate with elementary areas. Elementary areas correspond directly to their logical counterparts. The same analogy, as will be seen below, does not apply to composite areas. The model only deals with composition down to the elementary area level, the internal organization of which is governed by the appropriate object architecture. The notions of elementary and composite areas are similar to the blocks and frames defined in the Office Document Architecture standard [ECMA 1985], in which a more comprehensive layout model has been defined.

3.3.1 Elementary Areas

An elementary area is defined as one which holds a corresponding logical elementary object, and is associated with that particular physical architecture. An elementary area for an equation is thus based on the mechanism for composing expressions, while an image area would have a pixel-based model. Nested elementary objects (such as a table with embedded equations) would result in nested elementary areas accordingly. Although two nested areas may be associated with different physical schemes, the included area would merely be a black box with respect to the outer area.

Each elementary area therefore has its own physical model, of which a specialized formatter for the corresponding object must be aware. However, all elementary areas, irrespective of their internal architecture, are characterized by their box dimensions and a reference point, which would be set on the baseline if the object is formatted within ordinary text. Furthermore, at the lowest level of page description, areas with various architecture (for equations, tables, text block, line graphics, etc.) would eventually be resolved to a common set of primitives. An example would be the exchange of a document in its final form based on a page

description model like Postscript [Warnock & Wyatt 1982].

3.3.2 Composite Areas

As with logical objects, composite areas are collections of elementary and/or composite areas. An area for illustration, for instance, may have two elementary areas — *text* for the caption and *table* for the exhibit. However, except for floating illustrations, a composite areas does not necessarily correspond to a composite logical objects. This is because columns for flowing text seldom terminate at logical object boundaries (such as the end of a paragraph).

A typical page is composed of elementary text areas for running headers and footers, and composite areas for flowing text and floating illustrations (which are further decomposed). In simplified terms, a page composition process would first position floating objects, and then fill in remaining areas with flowing content. Further discussions on the layout process (with reference to the model described in this thesis) will be found in Section 3.4.2.

3.4 PROCESSING MODEL

We have discussed the static aspects of the document model. Let us now look at some activities for maintaining the logical structure as well as deriving the physical structure from it.

3.4.1 Editing Activities

We will first look at manipulation of the overall structure of a document, and then discuss maintenance of its generic structure (or architecture).

3.4.1.1 Global Structure Manipulation

As elementary objects would be maintained by the appropriate specialized editors, we only discuss here structure manipulation above the elementary level. We will identify the abstract operations in terms of two requirements.

- A. **Navigation.** As in any hierarchical editor, a user would be conceptually located at a particular node of a document tree. Here, the scope of selection would either be a composite or an elementary object, which corresponds to the subtree under the selected node. As will be seen in later chapters, the navigation commands defined below could apply to traveling within elementary objects as well.
1. **Default.** This would select the 'next' object according to a depth-first traversal of the document tree.
 2. **Root.** The entire document would be selected.
 3. **Arbitrary.** Any arbitrary object could be selected directly.
 4. **Sibling.** Provided the current object belongs to a sequence of elements, the next or previous sibling object would be selected accordingly.
 5. **Child.** Provided the current object is composite, one of its specified components would be selected.

6. **Parent.** Provided the current object is not the entire document, its parent object would be selected.
 7. **Follow link.** Provided the current object is connected via a non-hierarchical link, its sink or source object would be selected.
- B. **Object and Link manipulation.** These commands would maintain the fundamental as well as auxiliary structure of a document. Multiple selection of objects would be required for link manipulation.
1. **Create object.** This would substitute an empty slot of a composite object with a specified template which the model allows.
 2. **Create slot.** This would create an empty slot in a composite object, provided the latter has optional or self-repeating components, as defined in the model.
 3. **Remove object.** This would remove a selected object. If a compulsory field is removed, the deleted object would remain as an empty slot.
 4. **Create link.** This would establish a specified link between two selected objects. Syntax direction would be provided accordingly (§3.2.3.2).
 5. **Remove link.** This would remove a selected link between two objects. For chained objects, an alternative would exist for removal of the whole link structure at once.
 6. **Reorder objects.** This would enable arbitrary reordering of objects which belong to the same parent object.

3.4.1.2 Modification of Architecture

Modification of the generic structure (or model) of a document is needed to alter existing or introduce new definitions for composite object types. A special case would be definition of a document type, considering that a document is a composite object after all. Two approaches to specifying the syntax of document objects (namely, use of meta-language and graphical means) will be discussed in Section 3.5.2. We now focus on the abstract requirements for this purpose.

The definition of a composite object is normally based on other object types. The object being defined and those introduced in a definition are here referred to as the *basic* and *subsidiary* objects, respectively. Undefined subsidiary objects would themselves need to be defined. A document model is composed of a set of definitions, as illustrated by the production rules in Appendix A. The requirements for establishing and modifying a definition (or production rule) are identified as follows.

- A. **Define.** This would specify the relationship between the basic object and the subsidiary objects. Either one, two or an arbitrary number of subsidiary objects may be involved.
 - 1. **Set (sequence).** The basic object being defined as a set (sequence) of a specified number of subsidiary objects.
 - 2. **Self-repeating set (sequence).** The basic object being defined as a repeating set (sequence) of one subsidiary object.
 - 3. **Selection.** The basic object being defined as a selection from a specified number of subsidiary objects.

4. **Optional.** The basic object being defined as an optional instance of one subsidiary object.
 5. **Link.** The basic object (a link rather than an object identifier in this case) being defined as a non-hierarchical link between the two subsidiary objects.
- B. **Modify.** This would alter an existing definition either through insertion or removal of subsidiary objects, and only applies to selection, set and sequence definitions.

The meta-notation used to describe a document model is structured. This would enable the use of a structure editor to modify the model itself, just as a syntax diagram (which defines a programming language) may be viewed as an ordinary structured diagram. As will be discussed later, although the definitions of a composite object and its related subsidiary objects correspond to distinct production rules in the model, these could be specified as a single structure in a syntax diagram (as will be seen in Figure 3-5 later).

3.4.2 Formatting Activities

Due to our emphasis on logical organization in the model, instead of 'reinventing' some formatting procedures, we will only discuss briefly some layout issues with reference to the model we have defined. The discussion here is over-simplified when compared to a real formatting situation. We consider the formatting process at three levels. Figure 3-2 may be useful in the following discussion. The activities in the figure will be discussed further in the context of an abstract system (§3.5.2).

3.4.2.1 Elementary Object Construction

Elementary objects would be constructed by their specialized formatters, as will be discussed in their corresponding architecture in the following chapters. Except for flowing text, these objects could be formatted independently and presented to the second level of composition as black boxes. This process is represented by the 'elementary object formatter' in Figure 3-2.

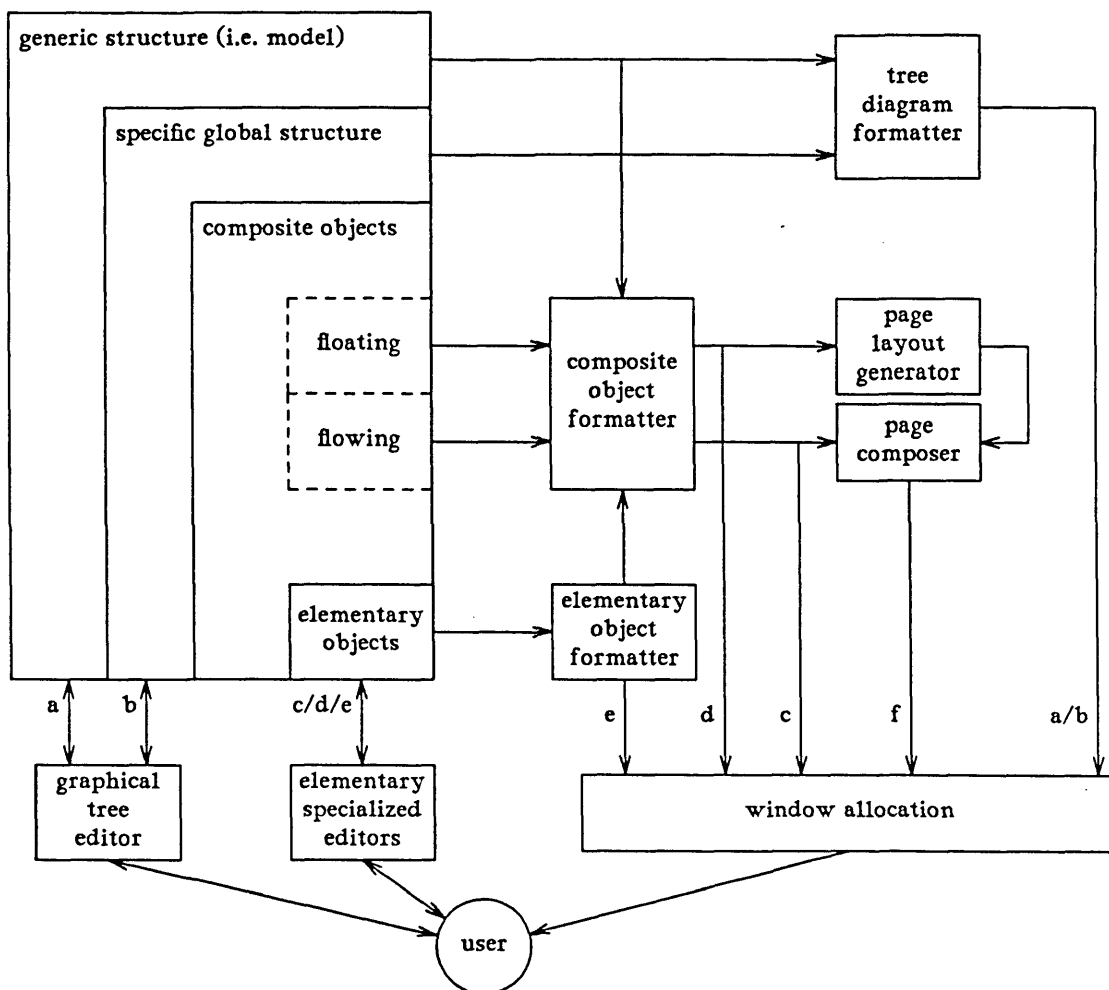
3.4.2.2 Composite Object Construction

Formatted elementary objects would be used to build composite objects, according to the associated style and formatting rules. For instance, the caption and exhibit of an illustration could be produced by the text and table formatters (as black boxes), and then positioned according to the style rules for 'illustration' objects. Only floating objects could be composed independently and prior to the page composition process. This process is represented by the 'composite object formatter' in the figure.

3.4.2.3 Page Composition

Independent floating objects such as illustrations would be produced as black boxes (just as elementary objects are hidden boxes to the composite formatter). Automatic positioning of these floating boxes would be a significant part of the layout process, and would be performed by the 'page layout generator' in the figure. Once this layout has been determined, the 'page composer' would simply fill in the remaining space with flowing content, producing the paginated view.

Overall Document Structure



Activities :

- a. generic structure (or model) editing
- b. global structure editing
- c. flowing content editing
- d. floating object editing
- e. elementary object editing
- f. paginated view

Figure 3-2. Overall structural model and related activities

A high degree of optimization could be possible in the automatic pagination

process. The decision on where to break a page, or the amount of interline adjustment to use, may influence the overall performance, such as the number of widows or orphans appearing in later pages. This is the motivation behind one optimal pagination algorithm [Plass 1981].

3.5 AN ABSTRACT SYSTEM

We now discuss a virtual system whose design could be based on the model we have just presented. First, we will look at the integration and structured editing aspects of the ‘could-be’ implementation. Next, we will discuss the various activities in a possible window-based implementation.

3.5.1 Integrated and Structured Editing

Let us look at how the abstract system would support structure editing and integration of specialized editors. A user could interact with a tree editor to manipulate graphically the global document *structure*, as depicted in Figure 3-1. Despatched under this supervising editor would be a crew of specialized editors which actually maintain the *content* of individual elementary objects, as illustrated in Figure 3-3. Structure editing would therefore be supported at various levels — the global document structure as well as structured elementary objects.

The ‘system’ would be fully integrated, without the need to import externally prepared objects. Suppose we are preparing the illustration in Figure 3-4, which includes an embedded table. While in the diagram editor, one should be able to

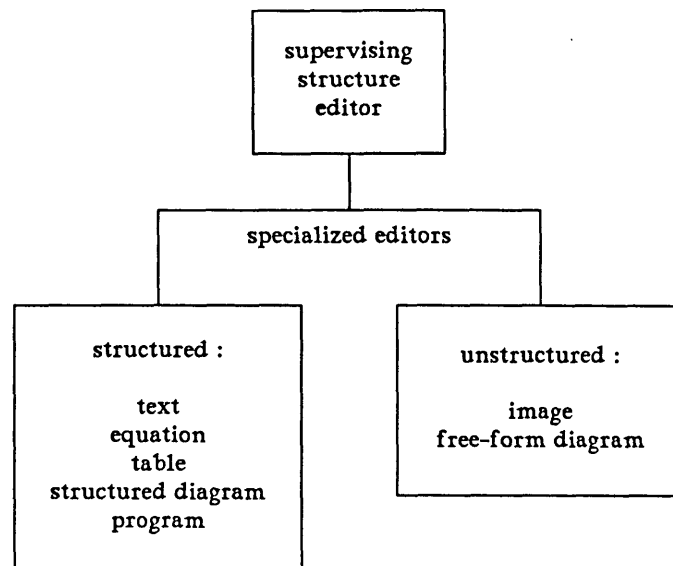


Figure 3-3. Supervising and dispatched editors

select the table. From then on, the specialized table editor would have taken over, and the diagram editor would be transparent until the former is deselected. Likewise, the equation editor could be invoked from the table editor. Specialized editors would therefore be *stacked* according to the particular structure of a document. For the sake of argument, suppose we need another column in the table to show the window functions graphically. The system should allow the user to reinvoke the diagram editor from the table editor. This would not be possible in Troff (with which this thesis is prepared) as diagrams cannot be nested inside tables. From the structural point of view, the equations are black boxes to the table, which itself is a black box to the diagram too.

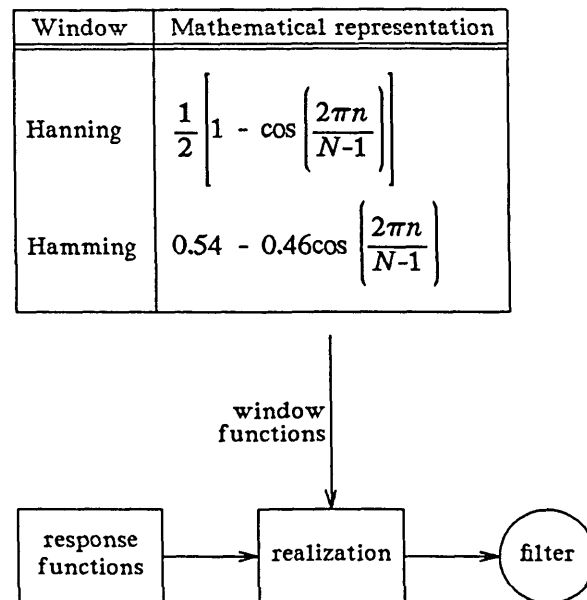


Figure 3-4. An example illustration

3.5.2 Window-based Activities

It has been argued that in the editing process, a user should not be distracted by the way lines are broken, or how floating objects are positioned. Instead, the user should only concentrate on the content and structure of a document. These defects, which are typical in interactive systems, have been referred to as a 'WYSIWYG syndrome' [Engelbart 1984].

The above argument may not be generally valid. Nevertheless, one of the main themes here is to support separation of structure and content manipulation. That is to say, apart from a page-oriented WYSIWYG mode, composite as well as elementary objects could be isolated for individual manipulation. Furthermore, there should be provision for modifying the global structure of a document

separately. With reference to the diagram in Figure 3-2, the following activities could be available in a window-based implementation.

- A. **Specific structure editing.** The global structure of a document would be maintained graphically. Structural relationships among objects, hierarchical or otherwise, would be visually presented, as demonstrated in Figure 3-1. This is particularly helpful when large documents are involved. Elementary objects would however be hidden as terminal objects, and be edited separately in WYSIWYG mode. The options defined in Section 3.4.1.1 would be appropriate for this purpose.
- B. **Generic structure editing.** The generic structure of a document is typically modified with the use of a meta-language [Quint et al 1986a] [ISO 1986] [ECMA 1985]. Alternative graphical approaches based on box diagram editing has been demonstrated [LeVan & Terreni 1986]. Like the specific structure, a document model could be specified graphically. The architecture of a document could be represented in a similar way as syntax diagram for programming languages (not a syntax tree, which is an instance rather than a model).

Since the tree now represents the model itself, instead of a structure conforming to it, a few extra requirements exist. A node in the specific structure corresponds to an *object*, while here it would represent an *object type*. Consequently, composition at tree nodes in the former case is confined to either sequence or set. An architecture tree, however, would also include grammar constructs at its nodes, namely repetition, selection and option. Figure 3-5 illustrates a definition of the object *chapter*. The activities discussed in Section 3.4.1.2 would be applicable here.

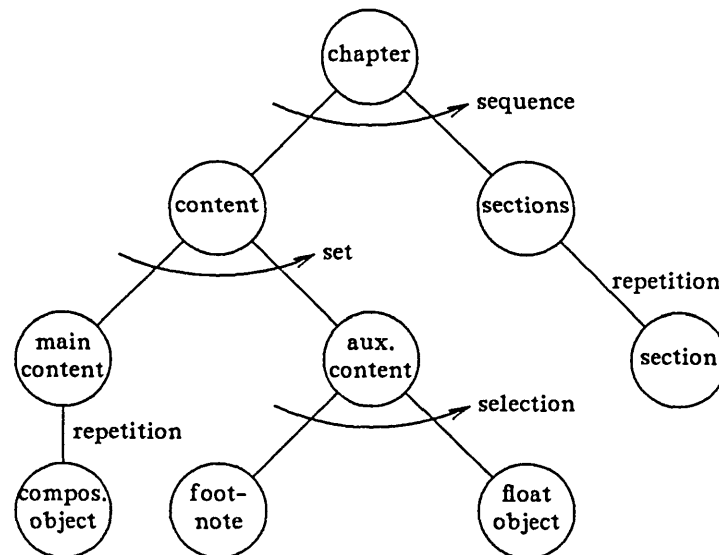


Figure 3-5. A demonstrative scene in model editing

- C. **Paginated view.** This would support the ordinary page-oriented WYSIWYG mode of editing, or provide paginated preview only.
- D. **Flowing content editing.** Unlike the paginated mode, flowing text would here be formatted to whatever window size is available; that is, non-WYSIWYG. Complex objects embedded in the running text could be represented symbolically and edited in a separate window, due to resolution problems.
- E. **Floating object editing.** Floating illustrations could be edited in WYSIWYG mode. Again, embedded complex objects could be isolated and edited separately, due to limitations on screen resolution.

- F. Elementary object editing. Elementary objects embedded in 'C' or 'D' could be isolated and manipulated in a separate window, which would be engaged with the corresponding specialized editor.

3.6 CONCLUSIONS

In this chapter, we have described the overall design of a document model, a formal presentation of which can be found in Appendix A. Special attention has been paid to the logical aspects of the model, and to the internal organization of elementary objects. The detailed architecture of three of the latter will be presented in the following chapters. We also discussed the abstract design of an integrated and structured system which could be based on the model. In the next chapter, we will present the architecture and an implemented editor for the elementary object *equation*.

Chapter 4

An Interactive Equation Editor

In this chapter we present the architecture of the elementary object *equation* as an example and describe an interactive editor implemented for equations. The term ‘equation’ is used here to refer to any mathematical expressions in general, instead of the particular class which implies an equality. The editor’s knowledge of the syntax of mathematical expressions makes possible structured and declarative manipulation. Furthermore, drawbacks present in most current equation editors have been eliminated.

4.1 INTRODUCTION

We will first identify the problems which are generally involved in the preparation of mathematically oriented documents. Then, we will review some important systems, discussing their philosophy and drawbacks.

4.1.1 Problems

High-quality scientific documents containing mathematics are produced successfully on popular systems like the EQN/Troff and the T_EX formatter. The degree of sophistication in computer composition of equations is evident in most mathematical journals. Weaknesses in these systems, however, lie in the specification process rather than the formatter’s capabilities. It is true that very complicated equations can be produced with professional quality, but it can be

tedious and inconvenient to specify and subsequently modify them. There are several areas which generally cause concern.

First, most scientific systems adopt a *procedural* approach to specifying equations, and the user indicates more or less precisely how to position individual symbols. This level of positioning control is not always desirable. A declarative approach, similar to that used in document specification [Reid 1980a], would be ideal for equations, as it would allow the user to interact in abstract and mathematical terms. For instance, when building a summation expression, one could just invoke a summation as an object, and then specify its various logical components. A procedural system typically needs to be instructed to put this above and that below the sigma sign, and so on.

The second concern is that a *language* is often used as the specification tool. Describing a two-dimensional graphical complex in a language which is linear is error-prone because the user must mentally correlate an expression's source description with its external appearance — the antithesis of direct manipulation. Invariably, the description is unnecessarily wordy. Apart from the symbols that actually appear in the equation, one has to include extra positioning commands as well as symbols to delimit physical units. As an example, the equation in [Knuth 1984, exercise 17-9], which has only 25 symbols, requires a T_EX description 117 characters long. Furthermore, modifying a previously entered equation is not trivial at all. In deciding on the necessary action, the user must analyze and then synthesize from the linear description.

The third weakness in most systems is *non-interactivity*, which means one cannot preview what is going to be produced without significant latency. We here distinguish non-interactivity from the previously addressed language issue,

because situations exist where a language source can be ‘instantly’ formatted for screen preview. The distinction were drawn between WYSIWYGery and direct manipulation in Section 2.2.1. In a non-interactive system, the debug cycle is frustrating when complicated expressions are involved, not to mention the consequent waste of resource.

Interactive and structured equation editors have appeared recently in the research domain, and have provided partial solutions to the above problems. Most of these systems lack a comprehensive model for mathematical expressions. As an example, the subexpression ‘ $\sin(\omega t)dt$ ’ of the integral $\int_0^{\pi} \sin(\omega t)dt$ is typically treated as an ordinary text string which is logically isolated from the integral sign itself. Consequently, the integrand ‘ $\sin(\omega t)$ ’ cannot be manipulated as an individual logical unit, nor is it known to be a trigonometric function. On the other hand, computationally oriented systems, such as problem solvers and algebraic packages, typically adopt a language-based and procedural approach. The next section presents a summary of existing facilities.

4.1.2 Existing Equation Tools

We will review some existing systems under three categories. The first two groups, namely, language-based and direct-manipulation tools, basically serve the purpose of document preparation. The third category of systems, most of which adopt the source-language approach, put more emphasis on computational than on typesetting aspects.

4.1.2.1 Language-based Tools

This category includes most conventional batch formatting systems, which inherently possess most of the deficiencies mentioned earlier. Previewing facilities for these formatters may help to provide a certain degree of ‘interactivity’, but the output is viewable only and the user still has to rely on the source text for modification [Trevorrow 1986] [Chen et al 1986b].

4.1.2.1.1 T_EX

We only discuss here T_EX’s handling of equations, which is a major integral part of the system [Knuth 1984]. Apart from its emphasis on quality, T_EX provides a high degree of user control in the composition process. T_EX is a procedural language. For instance, the input for the expression $\sum_{n=0}^M X_n$ is :

$$\text{sum } _ \{n=0\} \wedge M \ X_n.$$

A user thus selects the sigma sign and positions ‘n=0’ below and ‘M’ above it, without indicating to T_EX the expression’s logical structure. Furthermore, T_EX treats ‘n=0’ as an ordinary string, and is not aware that ‘n’ and ‘0’ are the index and lower bound of a summation. One therefore instructs T_EX how to position symbols, in contrary to the declarative approach, where a user would specify in logical terms and be freed from positioning details.

A T_EX specification can be unnecessarily wordy, especially when the user needs extra control like overriding the automatic scaling rules. The major weaknesses of T_EX are probably its unfriendly language and the lack of preview, which can be reflected by recent efforts in building interactive front-ends to the formatter. These systems can alternatively be viewed as exploiting T_EX as a postprocessing formatter for higher composition quality.

4.1.2.1.2 EQN/Troff

EQN is a preprocessor to the Troff formatter for handling equations [Kernighan & Cherry 1975]. Despite a comparatively lower degree of user control and composition quality, EQN is a more friendly language than T_EX. An EQN specification of the expression $\sum_{n=0}^M X_n$ is as follows :

sum from {n=0} to M X sub n.

Keywords like *from*, *to* and *sub* at least provide some hint regarding the expression's logical structure. The description is not declarative, however, because *sum* selects a sigma sign and not a summation expression, and it could have been any other symbol or unit in the above example. Similarly, *from* and *to* merely center anything below and above the previous unit (a sigma sign in this case) rather than implying the lower and upper bounds of some logical expression. EQN is a physically oriented language. The system has an underlying model which governs the construction of an equation. Constructs in this physical model correspond directly to the syntax definitions of the language itself.

4.1.2.1.3 Scribe

The original version of Scribe did not handle equations [Reid 1980b]. In the commercial Scribe, 'mathematical environments' have been introduced [Unilogic 1983]. While Scribe is one of the first truly declarative formatters, this objective has not been followed to a similar extent in handling of equations. A Scribe description of the expression $\sum_{n=0}^M X_n$ is :

Sum (From "n=0", To "M") X @Down (n)

The input is again oriented to the physical appearance of an expression. If one were to insist on the declarative philosophy with which Scribe handles text at a higher level [Reid 1980a], the system should provide, say, a 'summation environment', and be aware of its logical components and their relative positions (in much the same way as it associates other non-mathematical environments with typographic actions in the style database, as was discussed in §2.3.1.2).

4.1.2.1.4 PEN-MATH

PEN-MATH is an integral part of the PEN document system, one of the early hierarchical document editors [Allen 1981]. Its novel feature lies in its understanding of mathematical expressions, although quite limited when compared with problem solving tools which will be discussed later. Furthermore, it was one of earliest systems to demonstrate separation of content and style in formatting equations. Expressions are described in a language resembling the APL syntax, which consequently results in highly expressive and 'programmable' specifications.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

can be produced by entering

$$\sin(x) = '-+' / (x*1 + 2*0 i\{4\} \infty) \div ! 1 + 2*0 i\{4\} \infty.$$

Apart from the drawbacks inherent in language-based formatters, the APL syntax adopted by PEN-MATH may not be natural to casual users who are unfamiliar with the programming language.

4.1.2.2 Direct-manipulation Tools

We now examine a few interactive and structure-oriented editors for equations.

4.1.2.2.1 Levison's Editor

Based on a tree representation of equations, this editor provides the same degree of flexibility as hierarchical editors handle ordinary text [Levison 1983]. One browses through an equation tree with a similar set of commands used for navigating in a document hierarchy. However, cursor movement commands are closely bound to the internal tree structure instead of the external appearance of an equation. For instance, while at the lower limit of an integral expression, one issues a 'right' cursor command in order to move to the upper limit, just because the latter happens to be the 'right' sibling to the former in the internal representation! This to an extent contradicts the direct manipulation philosophy. Like similar tools to be introduced below, the logical structure of expressions is not adequately maintained.

4.1.2.2.2 Edimath

An interactive editor for equations developed at the IMAG laboratory in France [Quint et al 1986b], Edimath is part of a bigger project on an interactive and structured document system [Quint 1984]. Equations are modeled as a tree structure. Edimath provides nine types of nodes, each of which corresponds to a mathematical construct. The user describes an equation in a sequence which resembles writing it by hand, and is provided with appropriate syntax direction. For instance, when a summation construct is selected, the cursor moves automatically below the sigma sign to prompt further entry there, and then above the sigma sign, and so on.

The logical structure of an equation is not fully reflected in the internal representation, since the constructs provided are physically oriented. The summation construct, despite its name, actually implies any three individual units being composed vertically (the center one being a sigma sign by default). The summand itself is consequently not included in the 'summation' construct, and the index and lower bounds are not identified. Further discussion on this aspect will be found in Section 4.3.5.1.

4.1.2.2.3 Easy $\text{\TeX}$

Easy $\text{\TeX}$ is a scientific document system which aims at providing an interactive front-end to the $\text{\TeX}$ formatter [Crisanti et al 1986]. Easy $\text{\TeX}$ is very similar to Edimath, in its philosophy and in the available constructs, and consequently shares the limitations of the latter. The differences between Easy $\text{\TeX}$ (or Edimath) and the editor to be presented here lie basically in the extent to which equations are modeled logically and in the philosophy of the editor itself, and will be discussed in Section 4.3.5. Entry of equations using Easy $\text{\TeX}$ (or, in fact, Edimath) will be illustrated later in this chapter in Figure 4-14.

4.1.2.3 Computation-oriented Tools

We now turn our focus to facilities which are basically equation solvers rather than formatters. Notable examples include the MacSyma, Reduce and SMP algebraic packages. Problem solvers for personal computing include MathCAD, Eureka and TK Solver Plus.

All these tools have adopted the source-language approach for specification purposes. However, the extent to which the linear input format is 'visible' to the

user varies from one system to another. For instance, MacSyma explicitly maintains the source description of an equation even after it has been translated to a two-dimensional form of the expression. In this respect, MacSyma resembles a language-based formatter with screen preview (although a character-oriented terminal is normally used). On the other hand, although MathCAD also uses a linear input sequence, this format is incrementally (and 'instantly') interpreted upon entry to yield the graphical form of an expression, and is visually non-existent at any stage.

These systems normally support a rich set of mathematical functions and operators, and built-in plotting facilities. Since document presentation is not the primary concern, these tools typically provide less flexibility in the layout of equations. For instance, to indicate $\sum_{n=0}^M X_n$ using MathCAD, instead of creating a single expression, a user would require two separate statements, one to declare the range of the index and the other to specify an indefinite summation. The two generated expressions in this example are shown below, with the input sequence indicated in brackets.

$$\begin{array}{ll} n = 0 \dots M & (n : 0 ; M) \\ \sum_n X_n & (n \& X [n]) \end{array}$$

A recent project at Xerox PARC aims at an integration of four categories of mathematical software — numerical computation, mathematical typesetting, symbolic manipulation, and technical electronic mail [Arnon et al 1988]. As the objectives suggest, CaminoReal is a WYSIWYG, direct-manipulation equation editor with a limited set of built-in algebraic manipulation facilities. Access to external symbolic packages (such as Reduce, SMP and SAC-2) is provided via generation of output in the appropriate language.

Although the template-driven approach to equation specification in CaminoReal and that provided in the editor presented here are fairly similar, the two projects were developed independently. Furthermore, unlike CaminoReal, the editor presented here was built upon an explicit logical model for mathematical expressions (which is part of an integrated document architecture).

4.2 AN ARCHITECTURE FOR EQUATIONS

We present here a model for mathematical expressions. This forms the basis of an implemented equation editor which will be described in Section 4.3. The model has three main components — the *logical model* which formulates the mathematical structure of expressions, the *physical model* which generalizes the graphical or geometric composition of equations, and finally a set of *formatting rules* which specify the way each of the constructs defined in the logical model will be rendered geometrically based on primitives defined in the physical model.

4.2.1 Logical Model

Any equation, however complicated, has a hierarchical logical structure. This fact has led to previous implementations which model mathematical expressions as a tree structure [Arnon et al 1988] [Quint et al 1986b] [Crisanti et al 1986] [Allen 1981] [Levison 1983]. As discussed earlier, the extent to which expressions are modeled in most structured systems is inadequate for truly flexible manipulation. The work presented here is an extension in this direction, but aims at a more complete logical view of mathematical expressions.

A formal decomposition of the object *equation* can be found in Appendix B.1 where the notation introduced in Section 3.1.2 is used. This is actually a continuation of the overall document model presented in Appendix A, and is analogous to the syntax definition of mathematical constructs in a programming language. However, constructs supported in programming languages are relatively simple operators, while those appearing in scientific documents emerge from a much wider scope, ranging from complex integrals to matrix inversions. The model presented here, however, do not cover all possible mathematical notions. A discussion of its comprehensiveness will be found in Section 4.3.5.1.

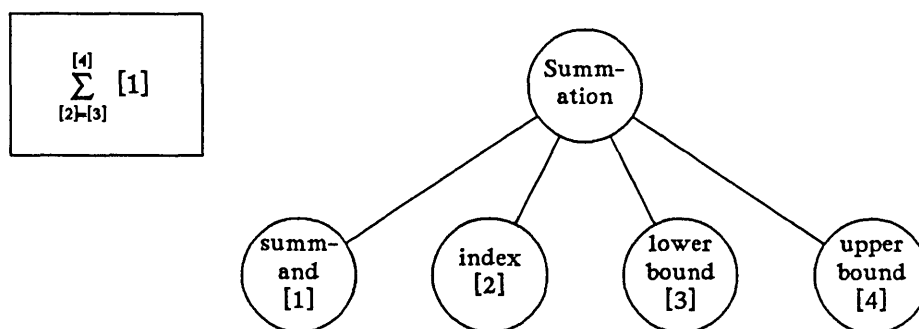


Figure 4-1. The logical composition of a summation

The model defines the syntax of equations. Each production rule either specifies a mathematical construct type or the composition of a specific construct. A standard summation, for instance, is composed of four logical components — the summand, the index, the upper and the lower bound, as illustrated in Figure 4-1 and by the following production rule :

$\langle \text{summation} \rangle ::= \langle \text{summand} \rangle + \langle \text{index} \rangle + [\langle \text{lower bound} \rangle + \langle \text{upper bound} \rangle]$

These four elements are further produced by other expressions, and are analogous to the expandable 'metanodes' in a structured program editor [Medina-Mora 1982]. Constructs may have a fixed number of subordinate elements, such as a unary reciprocal expression or a binary inequality. Alternatively, objects may have an arbitrary number of elements, such as matrices and vectors.

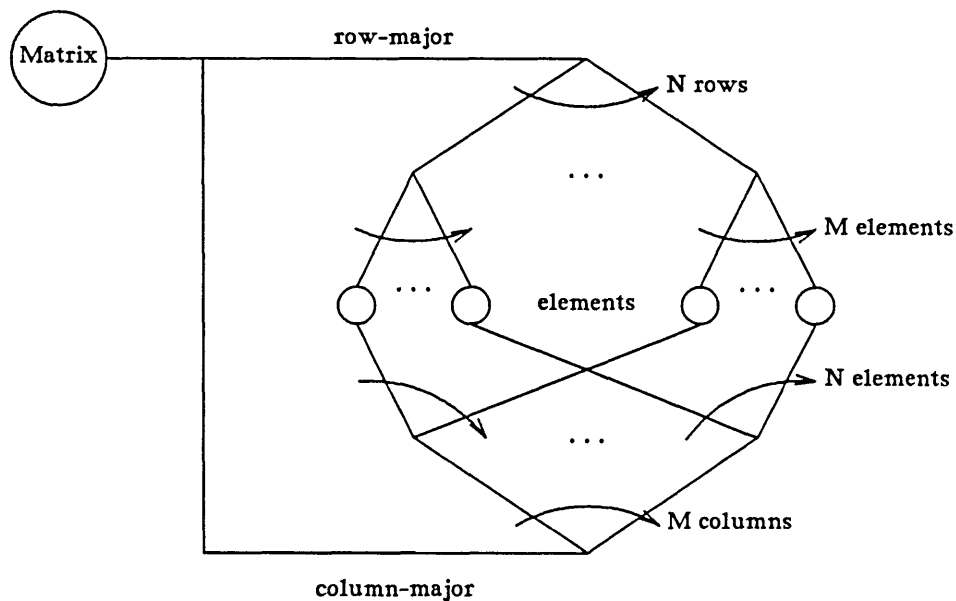


Figure 4-2. Dual representation of a matrix

The presentation in Appendix B.1 is slightly different from the conventional use of the BNF notation. A programming language definition would include the invariant syntax of a construct being defined, such as the reserved word 'while' in a *while-do* construct or the operator symbol '+' in an *addition* construct. Here, the composition of an object does not indicate the presence of any invariant content,

such as the sigma and equal signs in a summation, or the string 'lim' and the right arrow in a limit expression. These invariant symbols are however separately specified in the formatting rules (§4.2.3), and are inserted and positioned automatically in the formatting process.

A pure tree representation has been used for all objects except for constructs such as matrix and the like, or else these array structures would be dominant in either dimension. A row-major or column-major representation would result in inflexibility in manipulating the complementary dimension. A graph-like representation is used instead, as illustrated by an N-by-M matrix in Figure 4-2. Both rows and columns correspond to distinct nodes and thus exist as addressable units.

4.2.2 A Box-binding Model

The physical model for equations presented here is a relatively simple one, compared to say the math mode in T_EX. This is justifiable as we are more concerned about the logical organization of expressions rather than the composition quality. The model works satisfactorily for presentation on a graphics screen, as evident from an implemented editor which will be discussed later. Furthermore, the editor could have compiled its output in a postprocessing formatter language for even better print quality.

An equation has a hierarchical structure mathematically as well as physically. The expression in Figure 4-3, for instance, can be viewed as two horizontally bound boxes, the left of which can be further split into three vertically centered units. An equation is here modeled as a hierarchy of nested boxes. Except for terminal characters, any box is built from two or more sub-boxes according to a

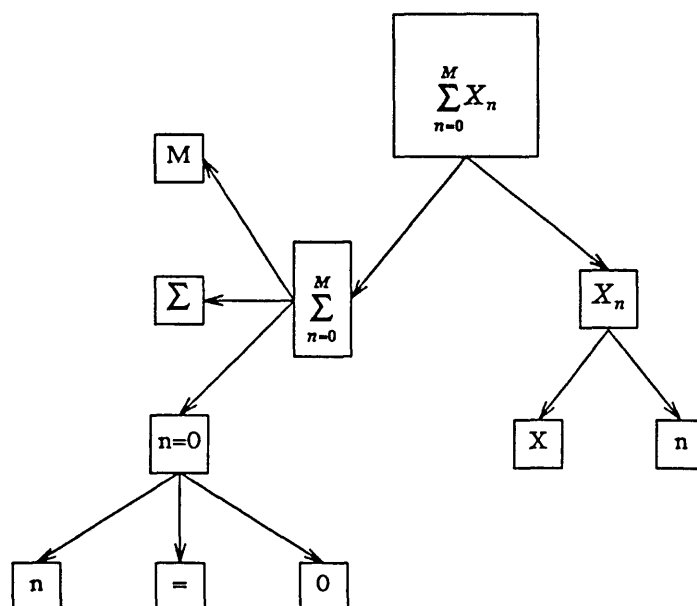


Figure 4-3. Physical composition of a summation

binding mode. This hierarchical composition implies a recursive formatting process which will be described in Section 4.3.4.2. The concept of boxes is borrowed from the layout model in T_EX. Here, a box is a rectangular region enclosing a portion of an equation, and could hold a single character, a subformula or even the equation itself. A box is characterized by its width, height, depth and a reference point which is normally aligned with the local baseline.

We have defined various binding modes, each specifying a way of constructing a box from two or more constituent boxes. The *vertical*, *horizontal* and *array* modes bind together an arbitrary number of boxes, while the rest accept a specified number of them. In the following discussion, one may refer to Figure 4-4 for the meaning of various typographical terms. Also, 'sub-boxes' refer to constituent

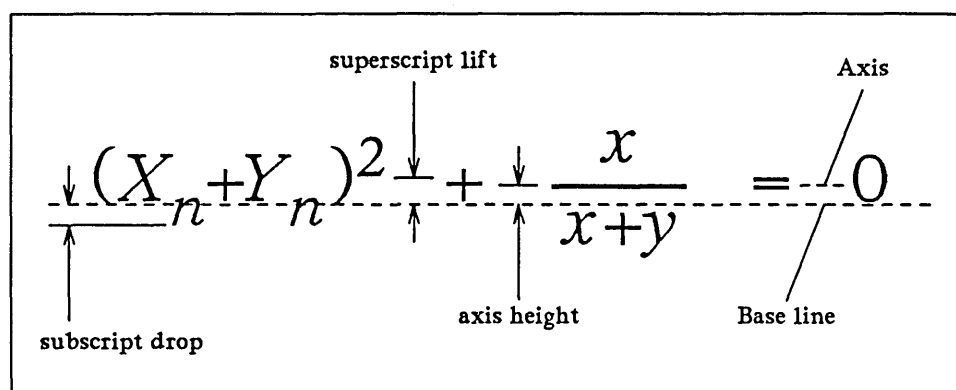


Figure 4-4. Terminology used in setting equations

boxes being bound together, while 'resultant box' is the one being constructed. A formal presentation of the model is found in Appendix B.2. Several of the binding modes we are discussing now are illustrated in Figure 4-5.

- A. **Over.** Two sub-boxes, a numerator and a denominator box, are centered vertically and separated by a horizontal bar long enough for the wider sub-box. The reference point of the resultant box is below the bar by the amount *axis height* (see Figure 4-4), irrespective of the size of the two sub-boxes, since the bar must lie on the *axis* in any case.
- B. **Superscript.** Two sub-boxes, a base and a script, are used to build the resultant box, whose reference point is that of the base box. There are two options. The script box can either be raised by a fixed amount (or *superscript lift*) regardless of the base box's height, or centered about the upper right corner of the base box. The latter is used when the base expression is not a

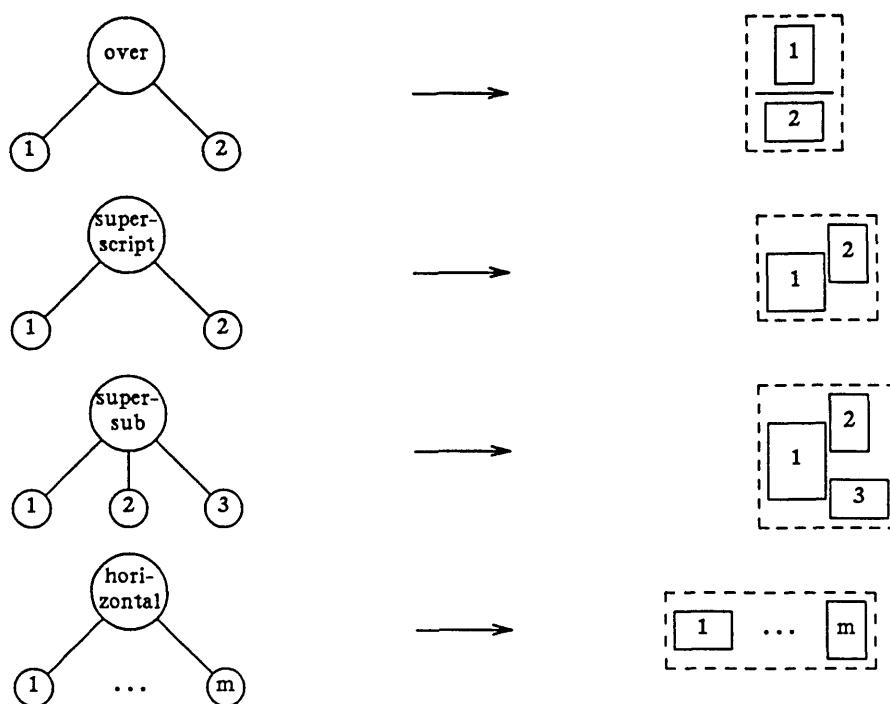


Figure 4-5. Some box-binding modes

single layer of symbols, such as matrices and fractions.

- C. **Subscript.** This is similar to superscript. The script box can either be lowered by a fixed amount (or *subscript drop*), or centered about the base box's lower right corner.
- D. **Supersub.** Three sub-boxes are used to build a simultaneous super-subscript box, whose reference point will be that of the base box. The two options of positioning the script boxes mentioned above also apply here. See Figure 4-5.

- E. **Pre-scripts.** These are similar to the three post-scripts just described (namely, superscript, subscript and supersub) except that the script boxes precede the base box instead.
- F. **Horizontal.** An arbitrary number of sub-boxes form a horizontal list, with their reference points set on the baseline. The reference point of the resultant box is that of the leftmost sub-box.

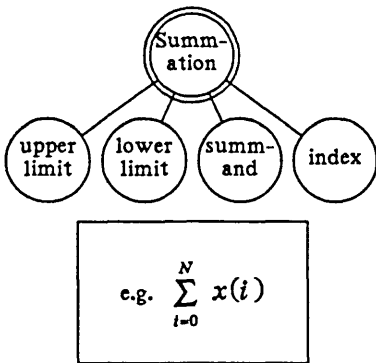
$$\text{---} \lim_{i \rightarrow 0} \text{---} \stackrel{def}{=} \text{---} \sum_{i=0}^N \text{---}$$

Figure 4-6. Three options in vertical binding

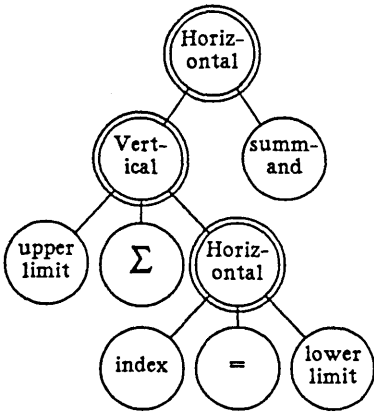
- G. **Vertical.** An arbitrary number of sub-boxes form a vertical list, which can be ranged left, right or centered (which is the default). The reference point of the resultant box can either be that of the topmost sub-box (if the *sink* option is selected), that of the bottommost sub-box (if the *float* option is selected), or that of a large-operator symbol (if one exists). These three cases are illustrated in Figure 4-6.
- H. **Array.** An arbitrary number of sub-boxes, each being a horizontal list of elements, are bound together vertically to form an array structure. Columns in the array are vertically aligned. Although this binding requires that the sub-boxes involved are horizontal lists, there is no restriction on the sub-boxes of these lists themselves, which are the actual elements in the array. The reference point will be mid way along the left side of the resultant box,

irrespective of its elements, since a matrix is normally centered about the *axis*.

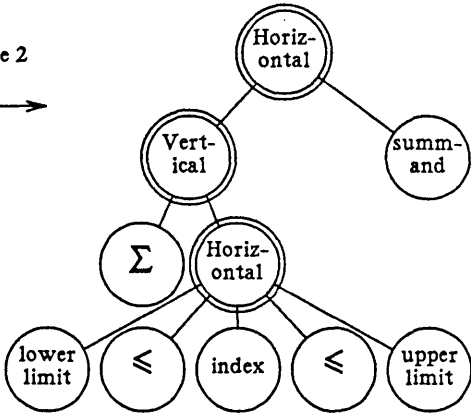
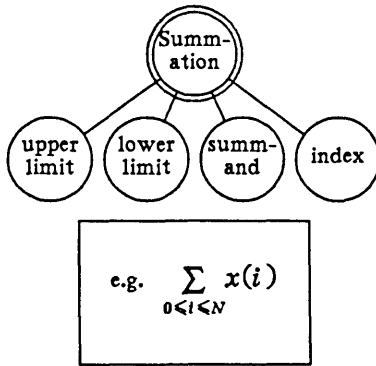
logical structure of a summation:



physical structure:



rule 1



rule 2

Figure 4-7. Two formatting rules for summation

4.2.3 Formatting Rules

We have already described the logical and physical models. As will be seen later, the editor interacts with a user to maintain the *logical structure* of an expression, which is then transformed to the corresponding *physical structure* according to a set of formatting rules. This mapping, which is transparent to the user, is part of the formatting procedures to be described in Section 4.3.4. These rules determine how each construct defined in the logical model is rendered physically. For each logical construct, a rule specifies the physical binding of its constituent components, any scaling required for each component, and the invariant symbols to be inserted.

In this way, declarative specification of expressions can be achieved. The logical model definitions and formatting rules already embody the syntactic and geometric information about standard expressions, respectively (which would otherwise be needed in a procedural system). For instance, a formatting rule for summations specifies the relative positions and automatic scaling down of various components, as well as the insertion of the sigma and equal signs (see Figure 4-7).

More than one formatting rule may be attached to a logical construct to accommodate content-style separation. Two expressions with an identical logical content may be presented in different styles, as demonstrated in Figure 4-7, in which two different rules are defined for the construct *summation*. So far, we have been talking about some rules for transformation in the formatting process. The actual logical-to-physical mapping of an integral expression which embeds a summation is illustrated in Figure 4-8. The mapping is based on the first formatting rule for summations (Figure 4-7) and another rule for integrals which is not shown.

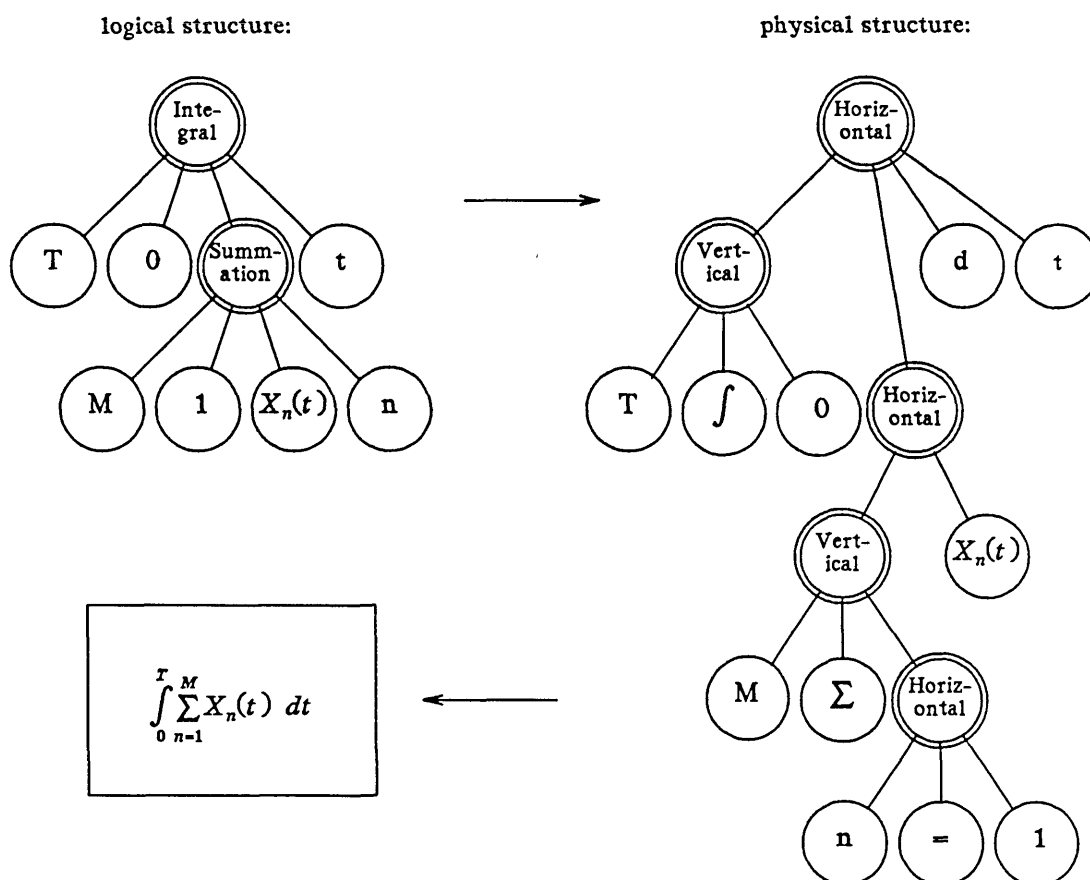


Figure 4-8. Logical-to-physical transformation of an expression

Some of the constructs in the logical model actually share a very similar transformation. For instance, most trigonometric functions basically just need a different function identifier to be inserted during the transformation (such as 'sin' for sine, or 'cos' for cosine). Only the formatting rules for those logical constructs which have been implemented as templates are listed in Appendix B.3.

4.3 THE EDITOR/FORMATTER

We now describe an interactive editor implemented on the basis of the model just presented. We will first look at the role played by various functional units of the editor, and then discuss the principles adopted in the user interface design. Next, we examine the approach with which the user manipulates equations, and discuss some formatting aspects of the system. Finally, we will evaluate the editor and the model which serves its design.

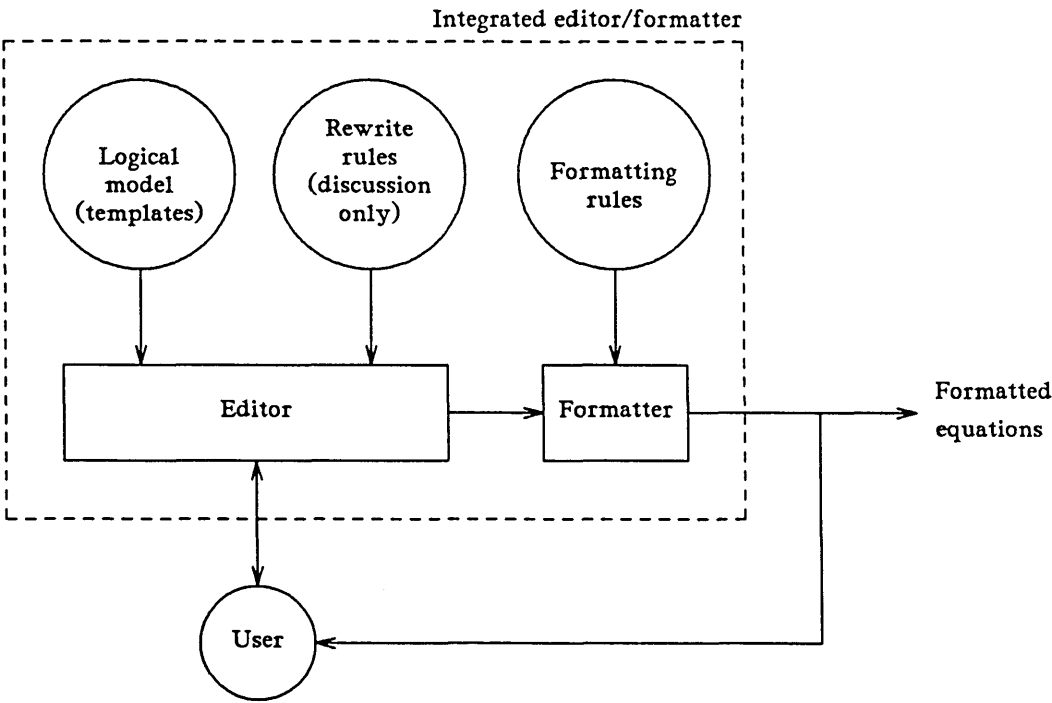


Figure 4-9. Functional blocks of the equation editor

4.3.1 Functional Blocks

This section provides an overview of the various units in the implemented system, the schematics of which are shown in Figure 4-9. More detailed discussion will follow in later sections. The two main components, the editor and the formatter, work in an integrated manner so that expressions are composed almost ‘instantly’ upon modification.

Common expressions selected from the logical model are built into the system in the form of mathematical templates. Equipped with this, the *editor* interacts with a user to construct and maintain the logical structure of an equation, which is then passed on to the formatter. By consulting the formatting rules, the *formatter* derives the physical structure of an equation from its logical structure, as the first step in the formatting process. The second computational step will be discussed in Sections 4.3.4.2.

The editor possesses only syntactic knowledge. Apart from a very limited set of simple calculation facilities, the editor is incapable of manipulating expressions symbolically. Let us look at the feasibility of implementing very simple rewrite facilities, based on the current architecture. By ‘rewrite’, we only mean some well defined and fairly inflexible symbolic substitution, instead of intelligent automatic reduction provided by algebraic packages. One example would be transformation between the following two expressions, based on integration by parts, where u and v can be more complex expressions.

$$\int u \, dv \rightarrow uv - \int v \, du$$

Transformations like this one would be useful, even though a user would have to specify the required transformation. One possibility would be to add a set of ‘rewrite rules’ to the model, which would specify various mappings between pairs

of logical structures (which, as in the above example, are semantically identical). These rules would be similar to the formatting rules except that the latter map between logical and physical instances instead.

Ideally, the end-user should be able to modify conveniently the database supporting the editor and the formatter, that is, the mathematical template library and the formatting rules. In the implemented system, the latter is transparent to the user. Rather than explicitly defined for each logical construct, a formatting rule is implied when the object in question is constructed as a mathematical template. Mathematical templates can be created using ordinary editing commands and appended to the library conveniently.

4.3.2 User Interface

As far as the user is concerned, an equation is here perceived as a nested graphical object whose hierarchy corresponds to its mathematical structure. As in any hierarchical or structure editor, a user can imagine that he or she is browsing through the equation tree and, at any time, situated at a particular node called the *cursor node*. The rectangular portion of an equation which corresponds to the cursor node is highlighted, and is here called the *box cursor*. This extended area cursor has appeared in most program editors [Medina-Mora 1982] and, in fact, some structured equation editors [Crisanti et al 1986] [Quint et al 1986b].

Instead of presenting the command list here, we only discuss some general requirements for navigating around an equation tree and in manipulating a selected subexpression. The implemented navigation and editing commands are listed in Appendix C.1.

4.3.2.1 Cursor Movements

The following are the basic requirements for cursor movement. Similar tree navigation activities have been defined at the document level (see §3.4.1.1). The discussion here is specific to selecting mathematical expressions.

- A. **Parent expression.** Provided the cursor node is not at the root, the user should be able to move one level up the hierarchy, and the box cursor extends to include its parent expression. This operation would eventually select the entire equation when applied successively.
- B. **Child expression.** Provided the cursor node is not at a leaf, a user could down the tree to one of the subordinate nodes. The box cursor shrinks and switches to the selected child expression accordingly.
- C. **Arbitrary selection.** This should enable a user to select any arbitrary subexpression or terminal symbol which bears no parental relationship to the current cursor. One condition, however, is that the target subformula must correspond to a node in the tree.
- D. **Entire equation.** This should provide a direct means of selecting the entire equation, instead of using the 'arbitrary' selection or even issuing several 'parent' selections.

4.3.2.2 Interaction Requirements

We have discussed the selection of an expression as the operand to an editing command. Let us now look at other aspects in planing the user interface. The following requirements have been met without any difficulties when building the

editor, via the use of a graphical user interface construction tool, namely, the GEM development toolkit.

- A. **Direct manipulation.** The implication of this objective goes beyond the direct interaction with the target equation being prepared. Direct interaction should be provided when appropriate, as exemplified in the editor by its graphical means of specifying matrices' dimensions (instead of numeral entry), or the use of a virtual on-screen keyboard containing mathematical and Greek symbols (which eliminates the need for typing in some odd codes). The latter example has appeared in the Star system [Smith et al 1982b].
- B. **Context-sensitive menu.** This requires that the command menu should be updated according to some context, which is considered at two levels here. First, this refers to the location of a user in the equation tree. As will be seen in Appendix C.1, some commands are valid only when a single symbol is selected while others apply to expressions only. Second, context refers to the state of an editing session, regardless of the cursor's location. For instance, when the buffer is empty, it would be sensible to hide the 'paste' command.

4.3.3 A Structured Approach

Let us now look at how two important concepts in structure editing, namely, template-based and structured specification, are employed in the editor. First, we introduce the facility of mathematical templates which the system provides, based on its mathematical knowledge. Second, we will discuss how the user, with knowledge of mathematics, goes about describing an equation to the editor. This structured approach distinguishes the editor from most interactive equation tools, with the exception of CaminoReal, which also provides mathematical templates

[Arnon et al 1988]. As mentioned earlier, however, the set of templates provided in the latter are not based on any explicit model within an integrated document architecture.

4.3.3.1 Mathematical Templates

The production rules in the logical model indicate the composition of mathematical constructs. The formatting rules, on the other hand, specify how these objects are rendered physically, which includes positioning, scaling and generation of invariant symbols. Based on these two pieces of information, the editor can deduce the standard form of various mathematical constructs. Some expressions have been selected from the logical model and built into the editor in the form of mathematical templates. A complete list of the implemented templates can be found in Appendix C.2, some of which are illustrated in Figure 4-10.

The empty boxes in a template correspond to its logical elements which are specific to particular instances of the expression. These unexpanded elements are here called *metaboxes* for convenience, and are analogous to the metanodes in structured program editors [Medina-Mora 1982], or the 'place holders' in CaminoReal [Arnon et al 1988]. Each metabox in a template can be replaced with a symbol, any expression or even another template.

Apart from those provided by the editor, a user can define additional templates which basically function as macros available in most formatting systems. Figure 4-11 demonstrates a user's definition of a sine-series template, and its subsequent usage where the metaboxes are filled using the 'paste' function.

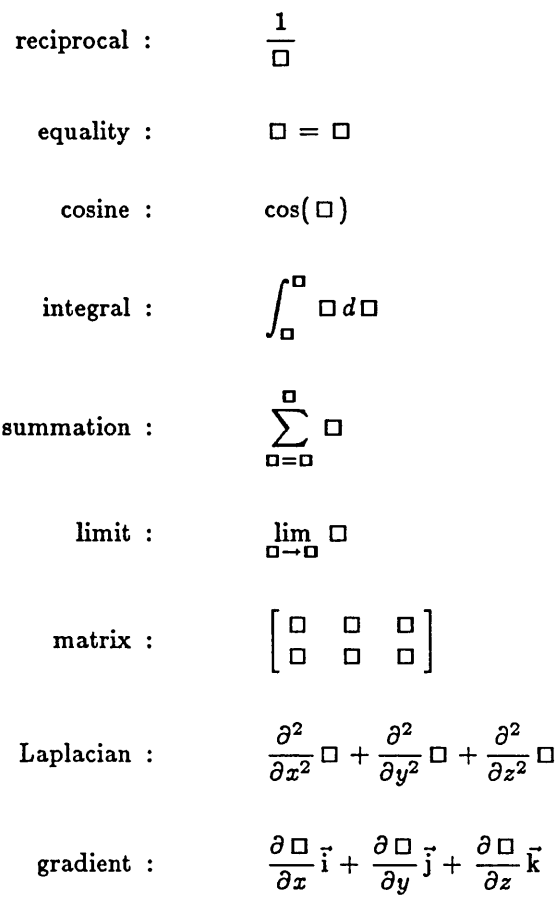


Figure 4-10 Some system-provided mathematical templates

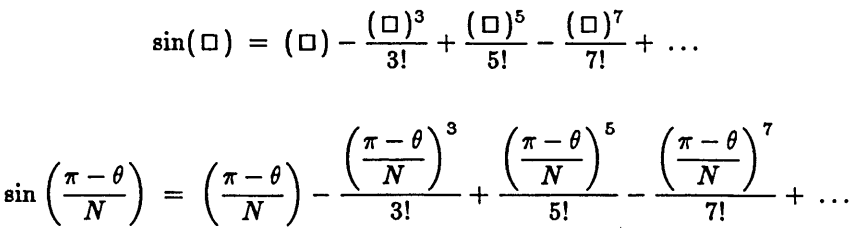


Figure 4-11 A user-defined template being expanded

Target equation :

$$\sum_{n=0}^M \frac{\lim_{x \rightarrow 0} \frac{\sin nx}{x}}{\int_0^\pi \cos^n \theta d\theta} > \prod_{n=1}^{2M} \frac{1}{n + \frac{1}{n + \frac{1}{n+1}}}$$

Sequence on screen :

$$\square > \square \quad (\text{inequality template selected})$$

$$\sum_{\square=\square}^{\square} \square > \prod_{\square=\square}^{\square} \square \quad (\text{summation and product templates})$$

$$\sum_{\square=\square}^{\square} \frac{\square}{\square} > \prod_{\square=\square}^{\square} \frac{1}{\square} \quad (\text{fraction and reciprocal templates})$$

$$\sum_{\square=\square}^{\square} \frac{\lim_{\square \rightarrow \square} \square}{\int_{\square}^{\square} \square d\square} > \prod_{\square=\square}^{\square} \frac{1}{\square + \square} \quad (\text{limit, integral and sum templates})$$

$$\dots > \dots$$

$$\sum_{\square=\square}^{\square} \frac{\lim_{\square \rightarrow \square} \frac{\sin \square}{\square}}{\int_{\square}^{\square} \cos \square d\square} > \prod_{\square=\square}^{\square} \frac{1}{\square + \frac{1}{\square + \frac{1}{\square}}} \quad (\text{ready for blank filling})$$

Figure 4-12 An initial editing session

4.3.3.2 Structured Specification

Using the templates just discussed, a user can specify an expression in a top-down manner according to its mathematical structure. Let us first consider the entry of an equation for the first time, referring to the example in Figure 4-12. The most abstract or concise description of the target expression is an inequality, so the user selects the appropriate template to start with. Since there is a product and a summation expression on each side, the user replaces the two metaboxes accordingly with the appropriate templates. Each substitution of a metabox reduces the level of abstraction. Furthermore, the editor displays a global view of the target equation at any stage, just as a program editor always presents the global form of a program.

This structured approach to entering equations seems to be a natural way for mathematically oriented users. Its suitability for other users may raise doubts, but we will leave this for a later discussion in Section 4.3.5. After an equation is created as illustrated above, subsequent modification to it is achieved via other structural commands listed in Appendix C.1. Figure 4-13 records a demonstrative session, continued from Figure 4-12, which illustrates the use of other cursor movement and editing commands.

4.3.4 Formatting Issues

We will first talk about the representation of equations in the editor, both internal and external, and then discuss the formatting process.

$$\begin{array}{lcl}
\sum_{n=0}^M \frac{\lim_{x \rightarrow 0} \boxed{\frac{\sin nx}{x}}}{\int_0^\pi \cos^n \theta d\theta} > \prod_{n=1}^{2M} \frac{1}{n + \frac{1}{n + \frac{1}{n+1}}} & \text{(initial state)} \\
\\
\sum_{n=0}^M \frac{\boxed{\lim_{x \rightarrow 0} \frac{\sin nx}{x}}}{\int_0^\pi \cos^n \theta d\theta} > \prod_{n=1}^{2M} \frac{1}{n + \frac{1}{n + \frac{1}{n+1}}} & \text{(select parent)} \\
\\
\sum_{n=0}^M \int_0^\pi \boxed{\cos^n \theta d\theta} > \prod_{n=1}^{2M} \frac{1}{n + \frac{1}{n + \frac{1}{n+1}}} & \text{(cut to buffer)} \\
\\
\sum_{n=0}^M \int_0^\pi \cos^n \theta d\theta > \prod_{n=1}^{2M} \frac{1}{n + \boxed{\frac{1}{n + \frac{1}{n+1}}}} & \text{(select arbitrarily)} \\
\\
\sum_{n=0}^M \int_0^\pi \cos^n \theta d\theta > \prod_{n=1}^{2M} \frac{1}{n + \square} & \text{(reduce to metabox)} \\
\\
\sum_{n=0}^M \int_0^\pi \cos^n \theta d\theta > \prod_{n=1}^{2M} \frac{1}{n + \boxed{\lim_{x \rightarrow 0} \frac{\sin nx}{x}}} & \text{(paste from buffer)} \\
\\
\boxed{\sum_{n=0}^M \int_0^\pi \cos^n \theta d\theta} > \boxed{\prod_{n=1}^{2M} \frac{1}{n + \lim_{x \rightarrow 0} \frac{\sin nx}{x}}} & \text{(select all)}
\end{array}$$

Figure 4-13 Subsequent manipulation session

4.3.4.1 Storage Formats

We will first look at the format of an equation maintained during an editing session, and then the minimal information needed to be stored externally.

4.3.4.1.1 Internal Representation

As indicated in Figure 4-8, the logical tree of an equation is structurally a 'subset' of its physical counterpart. The latter contains extra non-terminal nodes corresponding to physical but non-logical units (such as $\sum_{n=1}^M$) and terminal nodes holding invariant symbols (such as the sigma sign). A logical tree node corresponds to a mathematical subexpression while a physical tree node corresponds to any physical unit.

Due to their structural resemblance, these two forms of an equation share a single tree structure in the editor. This, of course, is an implementation choice. Let us look at the node attributes required in the internal representation. Those which are required in only one of the two forms (logical or physical) are indicated accordingly.

- A. **Math object (logical only).** This indicates the mathematical construct type of the corresponding expression (which is associated with the node in question).
- B. **Math field (logical only).** This indicates the field of its parent to which the corresponding expression belongs. For instance, an expression may be a summation (i.e. object attribute) while being the integrand (i.e. field attribute) of its parent which, of course, is an integral.

- C. **Binding mode (physical only).** This indicates the binding mode used in composing the corresponding box's subordinates, and applies to non-terminal nodes only.
- D. **Position (physical only).** This indicates the position of the corresponding box with respect to its parent (based on their reference points).
- E. **Dimension (physical only).** This indicates the dimension of the corresponding box, namely, its width, height and depth.
- F. **Delimiter.** This indicates the type of delimiter, if any, which encloses the corresponding unit. See Appendix C.1 for the available types.
- G. **Accent.** This indicates the type of accent, if any, which is positioned above the corresponding unit. See Appendix C.1 for the available types.
- H. **Shrink.** This indicates whether or not the corresponding unit is scaled relative to its parent. See explanation below.
- I. **Character code.** This indicates the symbol in the corresponding unit, and applies to terminal nodes only.

We have borrowed the scaling criterion from $\text{T}_{\text{E}}\text{X}$, which means that only three symbol sizes are used, namely, 'text', 'script' and 'script-script' (which normally correspond to 10-, 7- and 5-point). Here, if the *shrink* attribute is negative, the box in question is formatted using the same size type as its parent. Otherwise, its size type is one level below that of its parent; that is, 'script' if parent is 'text', and 'script-script' if parent is either 'script' or 'script-script'. The *shrink* attribute of a unit therefore affects all symbols within that unit. Relative scaling is therefore used, instead of storing the absolute size type, so that subexpressions are 'portable'

within the entire tree structure. Sub-expressions are rescaled accordingly when they migrate within an equation via, say, a cut-paste operation.

4.3.4.1.2 External Storage

Let us now discuss the external representation of an equation. The format described here is actually used to store ordinary equations as well as mathematical templates. An equation is saved as a pre-ordered list of node information. However, not all the data in the internal format described above need to be saved. The basic requirement here is that the stored information will enable the *logical* tree to be regenerated uniquely. Information which is specific to the physical structure is generated by the formatter instead. Only the logical content is therefore required, and attributes marked with 'physical only' in the previous section are not saved. As a matter of interest, a discussion on defining a standard for the external interchange of equations can be found in [Katz 1987].

4.3.4.2 The Formatting Process

The editor generates and maintains the logical structure of an expression. The physical form is then generated from that structure during the formatting process which has two phases. The formatter first derives the physical structure from the logical tree passed to it by the editor. This simply involves consulting the corresponding formatting rules, and is illustrated in Figure 4-8. The physical tree thus generated only indicates the binding modes in which boxes are to be merged together, but still lacks two pieces of information, namely, the relative positions and dimensions of individual boxes. These are computed recursively in the second phase which we are describing now.

Consider an elementary action, namely, binding together a number of sub-boxes according to a specified mode to form a resultant box. This should derive two quantities — first, the *dimension* of the resultant box and, second, the *positions* of resultant box's subordinates relative to itself. However, this requires that the dimension of the child boxes are available, and hence a box cannot be constructed until its sub-boxes are. At the lowest level in the hierarchy, the sub-boxes are actually individual symbols whose dimensions are readily available.

Apart from binding boxes together, another task of the formatter is to generate variable-size delimiters and accents. These big delimiters are built from concatenation of smaller 'fragment' symbols. For instance, the symbols 'l', 'l', 'l' and 'l' are used in building a left curly bracket of any size. Unlike ordinary symbols in the equation, however, these do not reside at the tree terminals. Instead, the delimiter type is stored as an attribute at the delimited expression's node, as discussed previously. The actual number of specific fragment symbols needed is determined in the formatting process. This is partly for the sake of a more compact representation, but mainly because the actual number of elements needed is unknown until the dimension of the delimited expression itself is determined.

4.3.5 Overall Evaluation

We first compare the way mathematical expressions are modeled here with the structure maintained in most editors. Then follows a discussion on the philosophy behind the editor itself as compared with similar systems.

4.3.5.1 The Model

With the exception of computation-oriented tools (which are typically language-based anyway), most interactive editors do not have an adequate logical view of mathematical expressions. The ‘summation’ construct provided in EasyTeX [Crisanti et al 1986] or Edimath [Quint et al 1986b], for instance, is defined as a vertical list of three units the middle of which is a sigma sign. It does not imply a complete summation since the summand (such as X_n in $\sum_{n=0}^M X_n$) is not even included, nor does it distinguish between the index and the lower bound which are both below the sigma sign. Similarly, these systems would treat ‘ $\cos\theta d\theta$ ’ in the integral expression ‘ $\int \cos\theta d\theta$ ’ as an ordinary string which is logically isolated from the integral sign, so that the integrand ‘ $\cos\theta$ ’ cannot be manipulated as a logical unit, nor is it known to be a trigonometric function. The constructs provided in these systems are rather physically oriented, and this has prevented truly logical manipulation.

We have distinguished between *physical* and *logical* constructs. The physical model only defines some general ways of joining together different boxes. The logical model, on the other hand, deals with constructs which carry some mathematical implication. For instance, although a fraction and a partial differential are both constructed internally using the *over* binding, these two are here distinguished in the logical model. A user of, say, Edimath would have used the physical construct ‘fraction’ in both cases, since there is no distinction between physical and logical constructs.

Two important pieces of information are embedded in the mathematical templates. Syntactic information indicates the composition of various mathematical constructs and the presence of invariant symbols. Geometric

information indicates any positioning and scaling requirements. The user therefore interacts in abstract and mathematical terms, and needs not supply the above information which the model already implies. This has resulted in a declarative specification which has previously been applied at a higher document level for describing non-mathematical materials [Reid 1980a].

The logical model we have defined is however not comprehensive. As it might be impractical to accommodate all possible constructs in various disciplines of mathematics, many of these are inevitably neglected here. However, our purpose is to demonstrate the abstract and structured approach to specifying equations, which should apply equally well to those 'missing' constructs had they been included in the model.

Even when the scope has been narrowed down to the areas we consider, there remains the question of deciding on the extent beyond which objects are not provided as primitive constructs. For instance, a Fourier transform could have been provided as a ternary construct that accepts the function to be transformed and the two domains involved. Instead, it is not provided in the model and has to be built from other primitives, namely, integral, multiplication and exponential function. However, high-level objects like this which the model does not provide can be constructed as user-defined templates, and would thus be as readily available as other templates provided by the system.

4.3.5.2 The Editor

The editor presented here has inherited the features of conventional structure editing (§2.2.3), namely, a top-down and template-driven approach. Furthermore, the global view of a target equation is always presented, with various parts

Target equation :

$$f(x, M) = \sum_{n=1}^M \frac{x^n}{n!}$$

Our editing sequence:

$\square = \square$	(equality template selected)
$\square(\square, \square) = \sum_{\square=\square}^{\square} \square$	(function and summation selected)
$\square(\square, \square) = \sum_{\square=\square}^{\square} \frac{\square}{\square}$	(fraction template selected)
$\square(\square, \square) = \sum_{\square=\square}^{\square} \frac{\square^{\square}}{\square!}$	(power and factorial templates selected)
$f(x, M) = \sum_{n=1}^M \frac{x^n}{n!}$	(blanks can be filled earlier)

EasyT_EX editing sequence :

$f(x, M) = \sum$	(normal typing until summation construct selected)
$f(x, M) = \sum_{n=1}$	('n = 1' entered followed by return key)
$f(x, M) = \sum_{n=1}^M \square$	('M' entered followed by return key)
$f(x, M) = \sum_{n=1}^M \frac{\square}{\square}$	(fraction construct selected)
$f(x, M) = \sum_{n=1}^M \frac{x^{\square}}{\square}$	('x' entered followed by selection of index construct)
$f(x, M) = \sum_{n=1}^M \frac{x^n}{\square}$	('n' entered followed by return key)
$f(x, M) = \sum_{n=1}^M \frac{x^n}{n!}$	('n!' entered followed by quit key)

Figure 4-14 Comparing with EasyT_EX

'hidden' as metaboxes at some stages (Figure 4-12). A user can therefore relate a partially built equation to its target form. At any stage, metaboxes can easily be identified as various logical units in the target equation, and templates are then selected accordingly. The task of specifying a complicated expression is thus broken into smaller subtasks. Figure 4-14 compares the ways an equation is entered using the editor here and using EasyT_EX (or, in fact, Edimath).

The global view also enables a user to develop an equation in any sequence. Consider Figure 4-15. After the matrix template is selected, the user may expand its four elements in any order (apart from the displayed sequence). A user of EasyT_EX or Edimath, on the other hand, has less flexibility in this respect when building an equation. As illustrated in Figure 4-14, these systems would more or less restrict the order of specification. However, these systems do provide a means which resembles very much handwriting an equation.

While mathematically knowledgeable users benefit from the structural approach, 'mathematically ignorant' users are not totally neglected. The *append* functions listed in Appendix C.1 can be used to compose an equation in physical terms, that is, without exploiting any templates. This unstructured means of entry may not be efficient, but at least leaves an option to those who fail to identify an equation's mathematical structure. Systems which are less insistent on a structured input sequence would be invaluable to these users who might prefer to 'copy' an expression instead, such as Edimath or EasyT_EX.

Although the editor presented here resembles structured program editors in some aspects, it does not provide any syntax direction which is a dominant feature in the latter. The equation editor does not guarantee that a mathematically valid expression is created. To a certain extent, an editor should not interfere with the

Target equation :

$$\mathbf{R} = \left(\det \begin{vmatrix} x & 1 \\ 0 & -\lambda \end{vmatrix} \quad \prod_{i=1}^N g(i, x) \right)$$

$$\lambda \quad \sum_{n=0}^M f(n, \lambda)$$

Sequence on screen :

$$\square = \square \quad (\text{equality template selected})$$

$$\mathbf{R} = \left(\begin{array}{cc} \square & \square \\ \square & \square \end{array} \right) \quad (2 - 2 \text{ matrix template selected})$$

$$\mathbf{R} = \left(\det \begin{vmatrix} \square & \square \\ \square & \square \end{vmatrix} \quad \begin{array}{c} \square \\ \square \end{array} \right) \quad (2 - 2 \text{ determinant template selected})$$

$$\lambda \quad \square$$

$$\mathbf{R} = \left(\det \begin{vmatrix} x & 1 \\ 0 & -\lambda \end{vmatrix} \quad \begin{array}{c} \square \\ \sum_{\square=\square}^{\square} \square \end{array} \right) \quad (\text{summation template selected})$$

$$\mathbf{R} = \left(\det \begin{vmatrix} x & 1 \\ 0 & -\lambda \end{vmatrix} \quad \begin{array}{c} \prod_{\square=\square}^{\square} \square \\ \sum_{n=0}^M f(n, \lambda) \end{array} \right) \quad (\text{product template selected})$$

Figure 4-15 Entering a matrix

logical validity of expressions, from the document preparation viewpoint. However, it might be desirable to provide a certain degree of syntax direction, such as the prohibition of equating a vector to a scalar expression. This is beyond the capability of the implemented system described here.

While the structural approach is generally desirable especially for complicated expressions, it is sometimes awkward to adhere to this view. For instance, consider the following equation :

$$((a+b) - c) = (x-y) * (p-q) .$$

The structural approach typically requires six template selections after which the metaboxes are filled in. It would be far more efficient to enter the equation simply as a text string. Although this is actually possible in our editor, those parts of an equation entered this way will lose their logical identity, which can only be recovered with some parsing scheme. In the present editor, structure manipulation is not possible with those parts which are not created in the expected structured manner. The point we are making here is actually the structure-flexibility tradeoff concerning, at a higher level, structured document systems. We will discuss this further in the concluding chapter.

4.4 CONCLUSIONS

We have described an interactive editor which has a more complete logical view of mathematical expressions than would be implied by pure text manipulation, as the design is based on an explicit model. This has resulted in a structured and natural way of manipulating complex equations interactively. It is simultaneously a

strength and a weakness since the mathematically literate user would find the editor natural while others might not be able to use it efficiently. It would seem that this is a stray movement towards authoring systems where the author is increasingly involved with tasks which were previously taken care of by the publisher, which might well include the keyboarding task. Based on the same argument, it would seem worthwhile to incorporate more specific author support in a technical document system. In the next chapter, we will introduce another elementary object — the table.

Chapter 5

The Elementary Tabular Object

In this chapter we present an architecture for the elementary object *table* as part of the entire document model. It is a generalized form of a relational data model from which some fundamental restrictions have been lifted to accommodate ordinary table structures which are not associated with any underlying database. The design of an abstract table editor will also be discussed.

5.1 INTRODUCTION

We will first give a more precise definition of the word 'table' used here, clarifying its relationship with several relevant concepts in data management systems. A brief introduction to the relational data model then follows. Finally, we will review some fundamental facilities for table manipulation.

5.1.1 Definitions

In the context of document preparation (or typographically speaking), tables can be defined as "rectangular arrays exhibiting one or more characteristics of designated entities or categories" [Beach 1986]. As an elementary object in the document model, *table* undoubtedly carries this implication. On the other hand, since it is basically a structured arrangement of information, the content of a table is often associated with some underlying data source, be it a database or some simple records. This accounts for our using a generalized relational model as a basis for

modeling the object. Let us now review several closely related concepts in the context of data management, and clarify their connection with the tabular object defined here.

- A. **Base table.** This refers to actual data entities which are stored physically in a database. Each row in a base table corresponds to a record which physically exists. Base tables are therefore a basic source of data.
- B. **Virtual table (or view).** This is a subset of data in the database (or base tables), extracted via a query language or a filtering process (as in forms processing systems). Multiple views can be created for a single base table. Alternatively, a view may span several base tables. Views are therefore 'derived' tables rather than a primary data source.
- C. **Form.** In forms processing systems, data is presented via structured forms which are stylized documents with a pre-defined field structure, very much like ordinary paper forms. Forms are therefore a formatted presentation of abstract views discussed above, and are maintained using a special form editor.

The relationship between the three is illustrated in Figure 5-1. Both base and virtual tables can be represented as tabular objects described in this chapter. On the other hand, a form is generally considered as a composite object introduced in Chapter 3, consisting of tabular and non-tabular fields. If a form contains a single tabular field, it is considered as an elementary tabular object instead. These correspond to non-tabular and tabular forms provided in Star [Purvy et al 1983].

The model to be described is based on a 'generalized' relational data structure, mainly due to the close link between relations and tables. The basic restrictions on

Name	Staff I.D.	Age	Rank	No. of children
Smith	1017	35	C	2
Jones	1213	42	B	3
Clark	1105	28	C	0
Brown	1515	45	A	4

a base table

Name	Staff I.D.	Age	Rank
Jones	1213	42	B
Brown	1515	45	A

a virtual table (or view)

Staff I.D. - 1213	
Employee Information	
Name	: Jones
Age	: 42
Rank	: B

a form

Figure 5-1. Relationship between a base table, a view and a form

a relation are relaxed, however, as will be discussed in Section 5.2.1.1. This is necessary since the flat table structure (as well as the other constraints) implied by the relational model would not be appropriate for describing general tabular materials. However, since the model is a generalization, it could still structurally describe strict relations as a special case.

Despite the structural compatibility between the tabular object and a relation, issues about linking the tabular model with an underlying database are beyond the scope of our work. In particular, the processing model here does not include the relational algebra component. Furthermore, there is no equivalent notion in our model for representing constraints on data values (as implied by domain in relations). Further discussions will be found in Section 5.4.

5.1.2 Relational Data Model

We will discuss the structural and manipulative components of the model separately. These are approximately analogous to the structural and processing elements in a document model, respectively.

5.1.2.1 Structural Components

Let us first define the fundamental terms. A *relation* consists of a *heading* and a *body*. The heading is composed of a fixed set of *attributes* each being associated with a certain *domain*. A domain is a set of permitted values of homogeneous type that an atomic (or non-decomposable) data entry can assume. The body is a time-varying set of *tuples*, each of which is a fixed set of attribute-value pairs, each value being taken from the domain associated with the corresponding attribute in the heading. A *primary key* is an attribute (or a group of attributes) for which, at any given time, no two tuples share the same values.

With reference to the relation represented by the base table in Figure 5-1, the top row is the heading which consists of five attributes. The attribute 'rank' would be associated with a domain, say, (A,B,C,D). 'Staff ID' is probably the

primary key and [Name-Jones, ID-1213, Age-42, Rank-B, Children-3] is one of the four tuples. A relation has several important properties.

- A. **There are no duplicate tuples.** This is because a relation is mathematically treated as a set of tuples which, of course, has no duplicate elements. This implies that a primary key always exists.
- B. **Tuples and attributes are unordered.** Again, the elements of a relation (i.e. tuples) or the elements of the heading (i.e. attributes) should not be ordered, based on definitions of set.
- C. **Attribute values are atomic.** If one perceives a relation as a table, this excludes repeating groups or hierarchical columns, and thus insists on a flat (or matrix-like) table structure. A relation is therefore always normalized.

5.1.2.2 Manipulative Components

This part of the model, often called relational algebra, defines operations applicable to one or more relations, creating another as output. These operations can be categorized into the following groups.

- A. **Select and Project.** These two operations extract some specified tuples and attributes, respectively, from a given relation.
- B. **Union, Intersect and Difference.** These generate a new relation from two input relations, applying the respective set operation to the two tuple sets.
- C. **Product and Join.** The former generates a relation from *all* possible concatenated pairs of tuples taken from two input relations. The latter is similar but produces only those pairs satisfying a certain condition.

- D. **Divide.** This accepts a binary and a unary input relation, and creates another consisting of all values of one attribute of the binary relation for which the other attribute's values match those in the unary relation.

5.1.3 Some Relevant Tools

Instead of reviewing most existing facilities for preparing tabular materials, we present here three tools which are representative. TBL is a batch but expressive tool [Lesk 1979], while TABLE is an interactive and fairly structured editor [Biggerstaff et al 1984]. Star, apart from providing fully structured table manipulation, also demonstrates integration with underlying records processing utilities [Purvy et al 1983].

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo- hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	11.1
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0

Figure 5-2. An example table

5.1.3.1 TBL/Troff

Due to the complex variety of styles involved in setting tables [Beach 1986], full user control can almost only be provided using a source language. In this regard,

TBL (a preprocessor to the Troff formatter) successfully produces complex tabular materials with flexible styles [Lesk 1979]. The table in Figure 5-2 is produced by the following TBL commands, where {T} denotes a tab character :

```
.TS
box;
cfB s s s.
Composition of Foods

.T&
c | c s s
c | c s s
c | c k k.
Food {T} Percent by Weight
{T} —
{T} Protein {T} Fat {T} Carbo-
{T} {T} {T} hydrate

.T&
l in in in.
Apples {T} .4 {T} .5 {T} 13.0
Halibut {T} 18.4 {T} 5.2 {T} 11.1
Lima beans {T} 7.5 {T} .8 {T} 22.0
Milk {T} 3.3 {T} 4.0 {T} 5.0
Mushrooms {T} 3.5 {T} .4 {T} 6.0
.TE
```

The row-dominance property indicated by the above specification implies that columns cannot be manipulated conveniently, especially when entries include complex content such as equations. Spanning is achieved quite 'procedurally'. The user would have to visualize a table's pre-spanned partitions, and then break each barrier accordingly. In the above example, one has to imagine that the table contains 9 by 4 compartments. The third line of the source — cfB s s s — merges four partitions (by spanning thrice) to produce a single compartment for the string 'Composition of Foods'. Should the user add or remove some columns at a later stage, he or she must make sure that all horizontally spanned entries still have the appropriate number of barriers broken.

Concerning object integration, only equations and pure text, but not diagrams, can reside within a table. This is because a source file would be scanned by the diagram preprocessor PIC prior to TBL. PIC will simply ignore any commands, PIC or not, that are enclosed within TBL delimiters.

5.1.3.2 TABLE

TABLE is an interactive and object-oriented tabular tool which supports structural manipulation [Biggerstaff et al 1984]. Several levels of granularity have been defined for selection of objects within a table, and cursor movements are context sensitive accordingly. For instance, while editing a textual table entry, the user moves within the string character-wise. At a higher level of granularity, the same cursor keys would instead help traveling among entries, columns or rows accordingly. TABLE is thus based on the notion of 'extended cursor'. Furthermore, organization of entries is not dominant in either dimension, and both columns and rows are easily accessible.

TABLE has two limitations. First, structured manipulation is not supported to a full extent. Taking Figure 5-2 again as an example, a user would first create a 9 by 4 grid which is actually displayed, and then issue repeatedly a special 'span' command to merge the compartments accordingly for the headings. This could be tedious if a table contains multiply nested columns, and subsequent modification to a table's structure is not natural. The second concern is TABLE's flexibility in integrating with other object types. Since TABLE actually uses TBL as a postprocessor, limitations suffered by the latter in this regard are inherited.

5.1.3.3 Star

Structural manipulation of tables is fully supported in Star [Purvy et al 1983]. One of the special objects provided by the system, table is treated as a hierarchy of fields which are organized in rows and columns. In Star terminology, 'field' refers to formatted data in a document and, in the context of table creation, correspond to table entries rather than table columns (which is more common). Columns may be divided and repeating, which correspond to nested columns and rows, respectively.

Forms processing systems typically provide complete database facilities, and put little emphasis on document processing functions. The records^{*} processing facility in Star, however, is well integrated with other document creation functions (including table manipulation). Records are presented as forms in print-quality, and are maintained using the familiar Star document and table editors. Tabular and non-tabular forms are distinguished in Star, although the latter usually contain embedded tables anyway. Mapping between data in a display document (i.e. form) and in the underlying records is achieved in a process called ladding, which provides the vital link between Star's document and records processing facilities.

5.2 AN ARCHITECTURE FOR TABLES

Like other elements in the document model, we consider the tabular object in terms of its static structure (i.e. the logical and physical components) and the associated processing activities.

5.2.1 Structural Components

Here, we will discuss the logical and physical models for the object. A formal presentation of the former can be found in Appendix D.1.

5.2.1.1 The Logical Model

Since the logical model is generalized from the relational data structure, it could describe instances of the latter. This implication of a possible link between the model and an underlying database is paralleled by the interchange of the *equation* object to an external algebraic package, as discussed in the previous chapter. However, some requirements must still be met before the model described here can handle strictly relational data. These will be discussed later in Section 5.4.

As mentioned earlier, the relational data model is not suitable for describing general tabular materials, due some of its fundamental restrictions. Let us now examine the requirements for representing ordinary tabular structures which necessitate relaxation of the relational restrictions (§5.1.2).

- A. **Duplicate rows.** While duplicate tuples are not allowed in a relation, there are no similar constraints on the rows in a table. In other words, primary keys do not necessarily exist in ordinary tables, although in most cases they do.
- B. **Row and column ordering.** While attributes and tuples in a relation are conceptually unordered, ordering of rows and columns in an ordinary table (as a document object) is often significant.

Table Heading			
Simple Field Heading	Simple Field Heading	Composite Field Heading	
		Simple Field Heading	...
...	...	...	...
...	...	Vertical spanning	...
...	...		...
...	...		...
...	...		...
Composite Record Heading	...	...	Entry
	...	...	...
...	...	Horizontal spanning	
...	...	...	...

(a) Terminology for tabular components

1		
2	3	
	4	5
6	9	12
7	10	13
8	11	

(b) Partitions in a table

Figure 5-3. Typical forms of a table

- C. **Non-atomic entries.** Data entries in a relation must be atomic, that is, cannot be relations themselves. However, hierarchically grouped rows and columns are common in ordinary tables containing nested sub-tables.
- D. **Spanned entries.** While entries in a relation cannot assume composite values, they cannot merge either. Ordinary tables however often contain spanned entries which are physically and logically merged.

- E. **Dual row-column representation.** The body of a relation is defined as a set of tuples which themselves are sets of attribute-value pairs. Relations are therefore inherently tuple-dominant. Since columns in a table can be nested, a column-major representation is also essential for selection of individual and grouped columns.

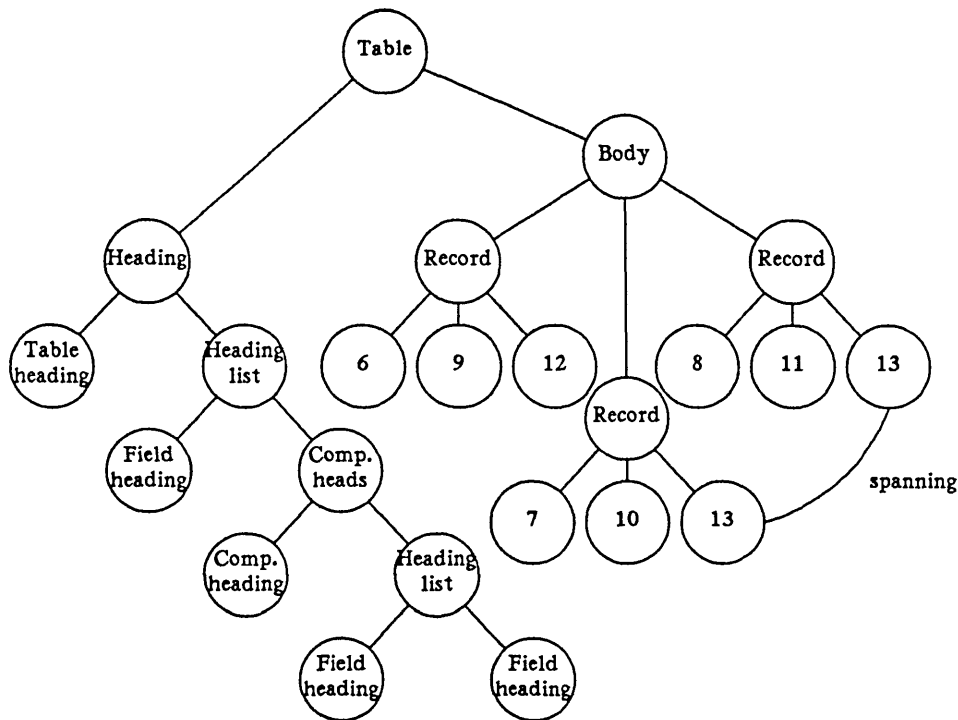


Figure 5-4. Record-dominant representation

With these requirements in mind, let us discuss the model in detail. It may be necessary to refer to the formal presentation in Appendix D.1. Figure 5-3a illustrates the terminology used here. A table can be viewed as either *record*- or *field*-dominant, both of which have a hierarchical structure. These two

representations of the table in Figure 5-3b are illustrated in Figure 5-4 and 5-5. The coexistence of these two representations would provide flexibility in addressing both records and fields, grouped or not (see 'E' above).

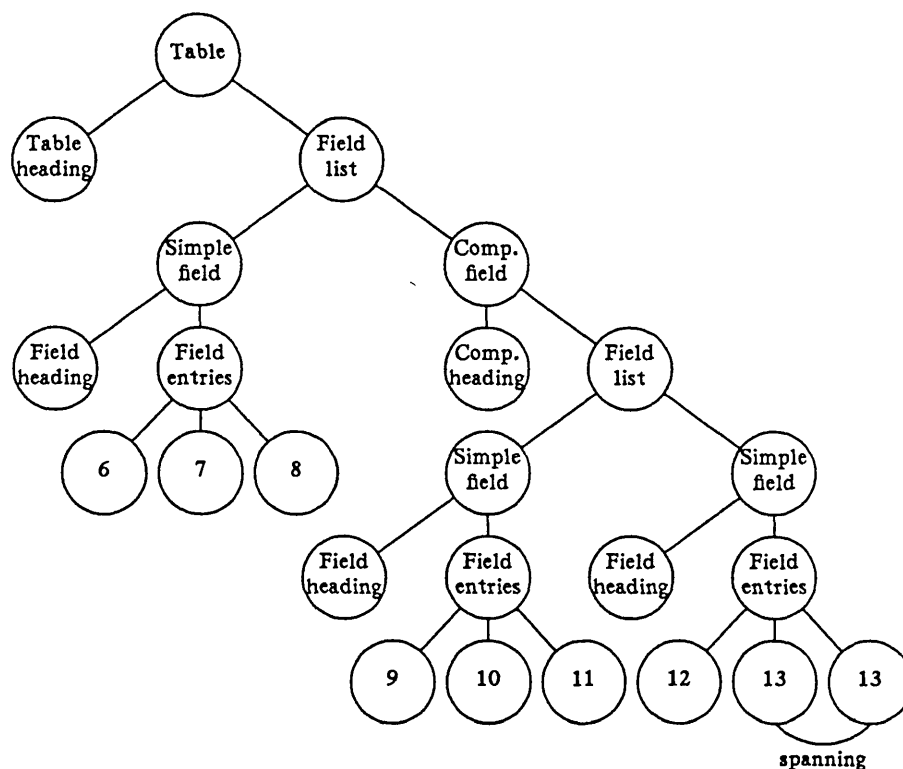


Figure 5-5. Field-dominant representation

Let us first consider the record-major representation (Figure 5-4), as it is more closely related to the relational structure (where tuples are dominant over attributes). A table consists of two major components — heading and body. These two, which correspond closely to their relational counterparts, are separated by a double rule in Figure 5-3a. Heading is defined here as a sequence, rather than a set,

of simple or composite headings (which are eventually decomposed to simple headings). The body section is a sequence of simple or composite records (which again reduce to simple records). Thus, records and fields are ordered (see 'B' above). Furthermore, composite records could be considered as tuples with non-atomic entries in a relation (which of course would no longer be a valid relation; also see 'C' above).

The field-major representation is relatively simple (Figure 5-5). A table is here viewed as a sequence of simple or composite fields. The latter eventually reduce to simple fields, which correspond to columns of entries in the body section as defined above. This representation therefore basically imposes a hierarchical organization on the columns of a table. This would enable selection of nested sub-tables, as well as declarative definition of a table's structure prior to data entry.

As far as the tabular structure is concerned, a tree 'terminates' at the nodes which correspond to table entries and headings. Taking the document tree as a whole, however, these 'terminals' of course lead to further subtrees of the appropriate elementary objects, just as the table structure itself can be embedded in a higher-level composite document object, such as a section. The logical model discussed here therefore serves as some intermediate structure in the entire document tree, and represents only the skeleton of a table, that is, excluding all entries. The two representations of the tabular structure coexist and share a common set of terminal nodes, resulting in a graph structure. A similar architecture has appeared earlier in another document model [Kimura & Shaw 1984].

Apart from the basic structure just described, two types of link are provided to connect table entries for the following purposes.

- A. **Spanning.** Two or more physically neighboring entries can share a single object, and are merged both physically and logically. Spanning can be horizontal, vertical or in both dimensions simultaneously. Only arbitrary spanning of data entries in the body section is represented in this way, as illustrated in Figure 5-4 and 5-5. Spanning for field headings or composite record keys is already implied by the hierarchy in the logical structure.
- B. **Data sharing.** Two or more entries, which may be physically remote, can share different instances of the same object. This is analogous to a similar link defined at the document level for sharing more complex objects (see §3.2.1.3). Entries linked in this way are only merged logically.

Each column (or simple field) is associated with an elementary object type. A field could thus be declared as a column of ordinary text, equations, diagrams, images, or even tables. The association of each field with an elementary object type could enable column-oriented special operations and automatic switching of editors when an entry is selected. These will be discussed in Section 5.3 in the context of an abstract table editor. This *field type* is analogous to the domain in relations. However, apart from identifying a column as, say, containing text, the field type imposes no further constraints on whether it is numeric or alphabetic, or on its data range. This is one of the 'incompatibilities' between the tabular structure and the relational data model.

5.2.1.2 The Physical Model

We now discuss a model which governs the physical composition of tables. Just like the box-binding model for equations, this model deals with construction of a target box from some 'terminal boxes'. For equations, these terminal boxes hold

some actual symbols, while here these correspond to compartments for table entries and headings.

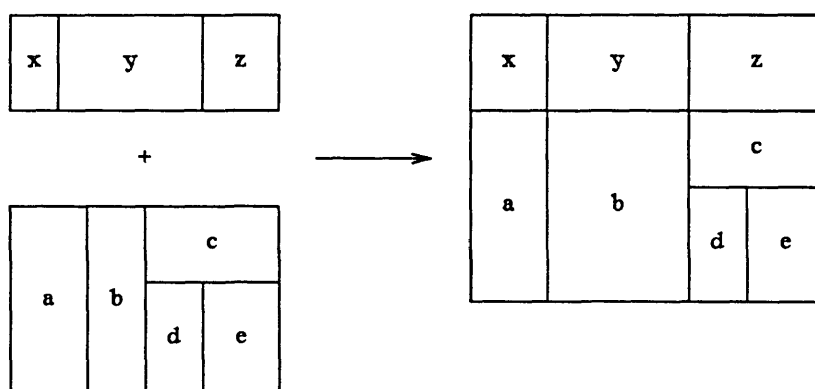


Figure 5-6. Scaling and alignment in composing sub-tables

It might seem that a subset of the box-binding model would suffice for building tables, since basically only horizontal and vertical bindings are needed here. However, tables are not treated here as a hierarchy of alternating horizontally and vertically bound boxes, due to the following requirements which do not arise in building equations.

- A. **Alignment.** Tables contain multiply nested sub-tables. At each level of composition, sub-tables are aligned in the appropriate dimension, as depicted in Figure 5-6. The three columns in this resultant structure may need further adjustment at a later stage. When sub-tables are being aligned, all the subordinate structures within them may therefore need to be rescaled and repositioned.

On the other hand, when subformulae in an equation are bound together, their subordinate boxes remain intact, in terms of size and relative position. Although matrices necessitate vertical realignment of elements when rows are being connected, the situation is much simpler due to the unnested flat structure involved. Furthermore, even when a matrix embeds another, elements of the inner matrix normally need not be aligned with elements of the external one.

- B. **Spanning.** Another requirement is the representation of spanned compartments. If a hierarchical model is used, provision is needed to resolve the ambiguity which arises when two sub-tables being bound together have different numbers of compartments at their interface (due to spanning). For instance, suppose the 'y' and 'z' compartments in Figure 5-6 were originally merged, it must somehow be indicated whether 'x' should be aligned with 'a' or with 'a' and 'b'. Merged compartments can of course be linked for this purpose, as in the logical model. This requirement does not exist in equations since spanning do not occur in matrices (or at least is not supported in our model).

Consequently, a flat array-like structure has been used in preference to the hierarchical model. A table is considered as a matrix of rectangular unnested boxes, which can be visualized by reinserting virtual partitions to all spanned compartments (that is, 'unspanning' the table). A table and its physical array are shown in Figure 5-7. Each compartment in the table has a unique identifier (Figure 5-7a), and can be associated with more than one boxes in the physical grid structure if it is a spanned entry. The physical model is basically a matrix of pointers to terminals in the logical tree (i.e. compartments of the table itself).

1		
2	3	
	4	5
6	9	13
7	10	14
8	11	
	12	15

(a) Compartments in a table

1	1	1
2	3	3
2	4	5
6	9	13
7	10	14
8	11	14
8	12	15

(b) The physical array

Figure 5-7. The physical array structure of a table

A similar matrix-like structure is used in the TABLE editor [Biggerstaff et al 1984]. However, since this matrix structure is the only internal representation, the logical structure is also oriented to the ‘pre-spanned’ condition of a table. Consequently, a TABLE user has to create spanned field headings procedurally, as discussed earlier in Section 5.1.3.2.

The need for alignment is implicit, since spanned objects reside in all columns or rows with which they need to be aligned, as depicted in Figure 5-7. Alignment for entries which are spanned in both dimensions (such as ‘L’ shaped compartments) are indicated in the same manner. As will be seen in Section 5.3.2, formatting

would proceed *independently* in both dimensions.

We have introduced the grid structure by referring to an already formatted table (Figure 5-7a). The formatter, however, would work the other way. The physical array would first be derived from the logical structure based on some formatting rules which will be discussed in Section 5.2.2.1. It is from this physical representation that the actual table is generated. This computational phase will be discussed in Section 5.3.2. These are analogous to the two formatting steps in building equations.

5.2.2 Processing Components

We first discuss the formatting rules used in transforming the logical tree to the physical array of a table. We then define a set of abstract operations for manipulating the logical table structure.

5.2.2.1 Formatting Rules

Like building equations, the formatting process would have two phases. The first stage, which we are discussing now, would convert the logical structure to its physical counterpart according to some rules which are listed in Appendix D.2. The record-dominant representation of the logical structure is used for this purpose, though the field-dominant form could have been used (but, of course, with another appropriate set of rules). The second phase, which actually computes a table's dimensions from this physical structure, will be discussed in Section 5.3.2.

Since both the logical and the physical structure of an equation are modeled as a tree, the formatting rules there basically specify transformation between specified

tree instances. Here, since the physical model is a flat structure, ‘tree-to-array’ conversions are involved instead. The rules in Appendix D.2 specify how each logical table construct, when identified in the structure, would be mapped to a physical array portion. The entire logical tree would be traversed, and the physical matrix structure would ‘grow’ as the rules are applied accordingly.

Two notations have been used in the appendix to denote concatenation of arrays, namely, *V-concat* and *H-concat*. The source array with the smaller dimension at the interface would be ‘extended’ to match the other. For instance, suppose array *A* contains a single element [1], and *B* is a row vector [2,3,4], a vertical concatenation of the two would yield the following array, showing an extension of array *A*.

$$\text{V-concat}(A,B) = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 4 \end{pmatrix}$$

As will be seen in Section 5.3.2, this array extension is responsible for generating multiple entries in the physical structure for spanned compartments. The physical structure in Figure 5-7b is generated by accumulating sub-arrays according to the rules while the logical counterpart is traversed (Figure 5-4).

5.2.2.2 Structure Manipulation

We now discuss some virtual operations defined for manipulating the logical structure of a table.

- A. **Modify.** The specialized editor for a selected entry (depending on the associated field type) would be invoked.

- B. **Span.** This would span selected entries in the body section of a table. The spanned terminal nodes in the logical structure would be linked accordingly.
- C. **Remove.** This would remove a simple field, a simple record, a composite field (with all associated columns) or a composite record (with all subordinate rows).
- D. **Insert.** This would insert a new field (or record) with respect to the selected field (or record), that is, at the same level as the selected construct in the nested table hierarchy.
- E. **Reorder.** This would be used to rearrange fields or records, which can be a combination of simple and composite elements at the same selected level.
- F. **Expand.** This would convert a simple field or record to a composite one. The former would be used in the initial definition of the structure of a table (§5.3.1.1).
- G. **Assign.** This would associate a simple field with an elementary object type, and is also used in the structure definition phase.

5.3 AN ABSTRACT EDITOR

We now consider a specialized table editor whose design could be supported by the model just presented. Before we discuss the user interface and formatting aspects, let us look at some requirements which are related to the objectives in defining the overall document model, and how these could be supported by the table architecture.

- A. **Declarativity.** Nested tables should be constructed in a structured way which eliminates procedural spanning of headings. The field-dominant representation would support this approach.
- B. **Object integration.** Objects other than textual data should be allowed in table entries. In this respect, the table structure defined here terminates with arbitrary types of elementary objects.
- C. **Special operations.** As each field is associated with an elementary object type, 'column-oriented' operations could be defined which are specific to the type of data involved.

5.3.1 User Interface

We now discuss two main activities, namely, the initial construction of a table structure, and the subsequent entry and modification of data. In database terminology, these would correspond to data definition and manipulation, respectively.

5.3.1.1 Structure Definition

This phase would include defining the basic structure of a table, specifying composite and simple field headings, and assigning object types to individual fields. This could be achieved with the *expand*, *modify* and *assign* operations (§5.2.2.2). Figure 5-8 illustrates the structure definition phase, in which field headings would be spanned automatically. The structure of a table thus created might need further modification (that is, redefinition). This would involve the operations *remove field*, *reorder fields*, and *assign*.

Departmental Composition		
Department	Student	Staff

A table with three main fields created.

Departmental Composition			
Department	Student		Staff
	Postgraduate	Undergraduate	

Centre field expanded to two sub-fields.

Departmental Composition					
Department	Student				Staff
	Postgraduate	Undergraduate			
		BSc1	BSc2	BSc3	

Structure of table fully defined.

Figure 5-8. Structure definition for a table

5.3.1.2 Data Manipulation

Once the structure of a table has been defined, a user might enter data to the body of a table. Operations needed for data entry would be *insert* or *expand* records, and *modify* entries (§5.2.2.2). Subsequent data maintenance might require *reorder* and *remove* records.

Let us consider the possibility of column-oriented operations for data manipulation. These would accept as input one or more fields of objects and

generate another, based on some operations which are specific to the associated field types. For instance, two columns of equations could be used to compute a new one, or two columns of images could be merged accordingly. The table editor would have to coordinate with the respective specialized editors for this purpose. For instance, column-oriented operations for equations would be provided by the equation editor itself.

5.3.2 Formatting Issues

Automatic layout of tables in a professional way involves many complex issues; a comprehensive study can be found in [Beach 1986]. The physical model defined earlier undoubtedly lacks provisions for many parameters required in a working formatter. We will only discuss the formatting process in the context of the physical model provided here, that is, computation of a table's dimensions, given its physical array (which would be derived from the logical structure as outlined in Section 5.2.2.1).

As seen earlier, each physical array element is associated with a compartment in the actual table (see Figure 5-7). Furthermore, each table compartment would hold a formatted elementary object, and is associated with the latter's minimal dimension. The problem here is to compute the dimensions of the physical array. Calculation in the two dimensions should be independent, even if there exist entries spanned in both dimensions. We will only consider calculation of the heights of rows.

We first consider the simple case where no vertical spanning is involved, such as the first and the fourth row in Figure 5-7b. The height of the row in question would be computed as the maximum height required by individual elements,

which corresponds to the tallest entry in that row.

On the other hand, rows which contain vertically spanned entries would be considered together, such as the second and third rows in Figure 5-7b. The bottom three rows would also be dependent, since the spanned compartments '8' and '14' overlap row-wise. Let us consider this latter example, and denote the computed heights for these rows as $H\text{-}tl$, $H\text{-}sl$, and $H\text{-}l$, corresponding to the third last, the second last and the last row. Also, we denote the height of a formatted object residing in the n^{th} compartment in the table as h_n . The problem is to compute the heights of these rows (i.e. the H's), subject to the following conditions.

$$H\text{-}tl > \max(h_7, h_{10})$$

$$H\text{-}sl > h_{11}$$

$$H\text{-}l > \max(h_{12}, h_{15})$$

$$H\text{-}tl + H\text{-}sl > h_{14}$$

$$H\text{-}sl + H\text{-}l > h_8$$

In general, there are two sets of conditions involved when determining the heights of dependently spanned rows. First, the computed height of each row must basically exceed the maximum height required by its entries, *ignoring* spanned elements. These correspond to the first three conditions above. Second, spanned entries must be accounted for, as illustrated by the remaining two conditions above.

We have so far ignored alignment of entries themselves across a row or a column. If entries are to be set on a common baseline, the heights and depths must be treated separately when determining the maximum required heights in a row. Similar requirements apply to the other dimension when, say, decimal alignment is

required.

5.4 CONCLUSIONS

We have presented a model for tabular materials which serves as an intermediate structure in the entire document hierarchy — leading to further elementary objects below, while itself being embedded in some higher-level composite objects (like floating illustrations). The model also supports structural manipulation which is one of the emphases in this thesis. The record-dominant representation of a table is structurally similar to the relational data model, except that the fundamental restrictions on relations have been removed.

Several modifications would be needed if the model is to be truly compatible to a relational structure. First, the constraints on a valid relation would have to be reinserted to the logical model. Second, the relational algebra component should be included in the processing model. Finally, the notion of domain would be needed in order that a table editor could validate entered data. In the model, each field is associated with an elementary object type, of which the *text* object is not further constrained.

In the next chapter we will examine the last of the three elementary objects which have been studied in detail — structured diagrams.

Chapter 6

A Structured Diagram Model

We finally come to the last of the three elementary objects examined in detail in this thesis, namely, structured diagram. We will justify the value of providing specialized drawing facilities in a general document system, and hence the need to incorporate that object in our model. While equations and tables both have a well defined scope within which their models are built, the object *structured diagram* covers a diversified class of structures. In this chapter, we investigate a 'generalized' model which is intended to support a subset of structured diagrams. The requirements for 'adapting' the model for specific diagram types (which conform to the generalized structures) will be studied.

6.1 INTRODUCTION

Let us first classify diagram preparation tools, in terms of the extent to which a system 'understands' the diagrams it creates, and exploits such knowledge accordingly. This 'knowledge' could be at a syntactic or even semantic level, as will be discussed in Section 6.1.2. Two categories have been identified, namely, general and specialized drawing facilities. We will then describe our objectives here in the context of providing the latter in a general-purpose document system.

6.1.1 General Drawing Tools

These systems have no knowledge about the logical syntax or implications underlying a specific diagram. Objects would be treated as geometrically oriented, and their logical significance, if any, would be disregarded. Due to the lack of an underlying logical model, a user would have to manipulate directly the physical layout of a diagram. These systems are therefore not based on the 'edit-format' philosophy, where a user would maintain the logical structure of a diagram and a *formatter* would derive its physical form automatically. Referring to Figure 2-1, these facilities are basically layout editors. We further divide this class into manual and constraint-based systems.

6.1.1.1 Manual Systems

A user of these tools would be responsible for deriving and maintaining geometric conditions in a diagram. Suppose the pattern on the right in Figure 6-1 is required, a user would have to ensure that a few conditions are always met; for instance, *af* aligned with *hk*, *ce* parallel to *dg*, *ab* of the same length as *cd*, and so on.

In a batch system, one typically maintain these relations using macros and registers. A typical example is the PIC preprocessor to the Troff formatter [Kernighan 1982], in which a certain degree of declarativity is provided in *positioning* control. For instance, the following command would draw a new box Box2, such that its upper left (or northwest) corner is situated midway the bottom (or south) of another box Box1.

Box2 : box with .nw at Box1.s

However, there are no similar facilities for expressing other constraints, such as producing a line of the same length as another, or being parallel to the side of an

arbitrarily oriented box.

An interactive system often provide simple facilities such as alignment grids for maintaining geometric conditions, but of course not to the extent achieved in constraint-based tools which will be discussed in the next section. Most modern object-oriented graphics editors (such as MacDraw and the like) enable a user to organize objects hierarchically, and compose complicated diagrams in a structured manner. This organization is basically geometrical, however, and the underlying logical syntax of a diagram is not taken into account.

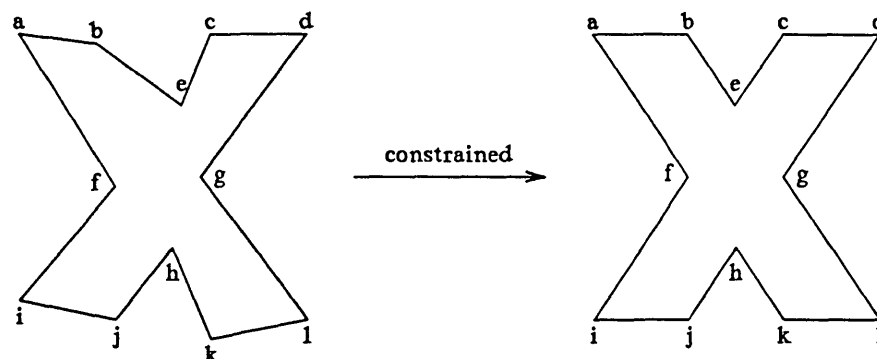


Figure 6-1. Applying constraints to a pattern

6.1.1.2 Constraint-based Systems

We now introduce a class of systems which allow a user to specify geometric relationships among picture elements declaratively via a set of constraints, which are usually satisfied using numerical methods. The types of constraints supported vary among systems, but typically include making lines horizontal, vertical,

parallel, perpendicular, of equal length, or forcing points to lie on a line or a curve. Constraint-based systems include Sketchpad [Sutherland 1963], Metafont [Knuth 1979], IDEAL [van Wyk 1981], Juno [Nelson 1985] and ThingLab [Borning 1979].

Two basic steps are typically involved in defining a diagram. Consider the target pattern on the right in Figure 6-1. In an interactive system, a user would first create an approximate profile (or sketch) of the target diagram, as depicted by the left pattern in the figure. The second step would be specification of constraints on the sketch by interactive means, as in Sketchpad, Juno, or ThingLab. In a batch system, constraints would typically be specified in terms of a set of equations, while the sketch would be produced by some conventional drawing commands, as in Metafont and IDEAL.

As will be seen later, constraint-based graphics tools could be useful in the specification of formatting procedures, when we discuss adapting the model presented in this chapter for specific diagrams.

6.1.2 Specialized Graphics Systems

The tools we have described so far manipulate the layout of a diagram directly. We now consider systems which possess knowledge specific to the diagrams they handle. These are basically CAD tools (such as software engineering design systems) which have expert knowledge of a specific class of diagrams. The knowledge of diagram syntax or even semantics (as will be distinguished below) enables specialization in the following ways.

- A. **Structure Editing.** Instead of treating a diagram ordinary graphics data structure, these systems are aware of the underlying 'diagram syntax' which represents logical relationships among elements in the specific diagram. These editors would interact with the user in logical terms and provide syntax direction to ensure the validity of a diagram.
- B. **Analytical power.** Structural manipulation just mentioned can be supported by syntactic knowledge alone. Diagram analysis, on the other hand, would necessitate semantic knowledge. This function is only provided in very specialized applications such as specific design tools, where document generation is only a secondary concern.
- C. **Automatic Layout.** Based on the 'editor-formatter' philosophy, the user would only maintain the logical structure of a diagram, while the system would derive its physical layout automatically. The specialized formatter would understand the specific rules and requirements in presenting the logical content.
- D. **Content-style separation.** Formatting style applies not only to textual content, but also to graphical illustrations (with the same advantages as in the former). Separation of content and style specification in a computer graphics system has been demonstrated [Beach & Stone 1983]. This is only possible in systems which maintain the logical structure of diagrams.

6.1.3 Objectives

The degree of specialization of graphics editors more or less depends on the scope of diagrams that can be handled, and we have above identified two classes of systems

approximately at the extremes. At one end, we have highly knowledgeable systems which only handle diagrams from a very narrow domain. These systems, which have a thorough understanding of the underlying diagram semantics, function primarily as design rather than drawing tools. At the other end, general-purpose editors have very limited or no knowledge about the underlying syntax or semantics of diagrams. Two examples which have aimed at a specialized system but yet accommodate a variety of diagrams include Rainbow, a multi-purpose CAD tool for network-oriented systems [Wiseman et al 1972], and MACCAD, an specialized tool for software engineering related diagrams [Jones 1986].

As one of the objectives in defining the document model is to incorporate structural (or syntactic) knowledge of specific objects, logically oriented diagrams should not be neglected, despite the diversity involved. However, diagram semantics are not accommodated in the model. It seems impractical for a general-purpose document system to integrate the analytical power offered by various specialized design tools. Consequently, as far as diagram preparation is concerned, the model should support structure manipulation, that is, specialized at the syntactic level only. In the next section, we will study a generalized model which is intended to describe a subset of logically structured diagrams.

6.2 A GENERALIZED MODEL

We have already outlined the purpose of including syntactic knowledge of specific diagrams in the model. However, the diversity of diagrams involved has prohibited the provision of each specific diagram as an elementary object type. We will study a generalized model for a limited set of structured diagrams. The model

is intended to be 'adaptable' for specific diagrams. No implementation is available to assess the feasibility of customizing the model to produce a really 'specialized' editor. However, requirements related to this issue will be discussed in Section 6.3.

The basis of the model is the definition of *diagram class*, which encompasses a set of diagrams having a similar underlying logical structure. Diagrams belonging to a particular class would share an *abstract model* and a corresponding *virtual editor* which would maintain the associated abstract structure. This is similar to the notion of 'virtual editing' in multimedia document manipulation, in which a set of abstract and universal editing operations are identified to maintain a diversified class of objects such as voice, image, or text.

We first discuss the diagram classes provided in the model, and then study the virtual operations required for each class. Finally, we will discuss the procedures involved in building specific editors for diagrams which, of course, belong to one of the classes supported.

6.2.1 Diagram Classes

Three classes of diagrams are identified here; these are namely tree-based, network-based and data-driven diagrams.

6.2.1.1 Tree-based Diagrams

This class includes hierarchically composed diagrams which can be perceived as a tree structure. Some examples are flow charts, syntax diagrams, parse trees, structured charts, decision trees, pedigree, and even mathematical expressions (if treated as a graphical object). Diagrams which are basically graph oriented but

hierarchically decomposable locally, such as data flow diagrams, belong to the network class to be discussed next. The generic structure of this class of diagrams is summarized below. This object exhibits a close similarity to composite objects defined at a higher level in the document hierarchy.

- A *diagram* is a hierarchy of nested *objects* or sub-pictures. An object corresponds to a node in the diagram tree, and is an instance of some object type.
- An *object type* definition specifies the valid composition of an object instance, namely, the permitted number and types of subordinate objects. These definitions form the basis of incorporating specific diagram syntax in the model.
- Nodes in a diagram tree, terminal or not, are associated with elementary objects which constitute the basic content of a diagram. These contents are typically elementary 'text' objects, and may reside in terminal nodes only (as in equations), or in non-terminal nodes as well (as in organizational charts).
- Non-hierarchical *links* are defined to connect objects of specified types. These can be associated with data which is typically textual information to displayed alongside the links.

Consider as an example the model for syntax diagrams (which represent a language's syntax definition graphically). The object types involved would be repetition, option, selection and sequence. The definition of, say, the type 'repetition' would specify that only a single subordinate object is allowed (which can be any of the four types except 'repetition' itself). The data associated with terminal objects in this case would be the reserved words and permitted symbols of the language being defined. This model also fits easily the definition of

mathematical expressions if one treats them as graphical objects.

6.2.1.2 Network-based Diagrams

This class of diagrams are perceived as a graph structure (directed or not), that is, being composed of distributed and interconnected elements. Examples include data flow diagrams, finite state transition diagrams, circuit diagrams, signal flow graphs, and network configuration. We now discuss informally the generic structure for this class.

- A *diagram* is a graph of interconnected *objects* which are instances of some defined type. Objects correspond to nodes in the underlying graph and are connected via links (or edges in the graph).
- An *object type* definition would specify the types of objects and the permitted number of (incoming and outgoing) links that can be connected to its instance. This would introduce specific diagram syntax to the model.
- Objects and links are associated with elementary objects which again correspond to the basic content of a diagram. These contents are typically ordinary ‘text’ objects but can be others like equations (as in a system flow diagram).

Consider a data flow diagram model, five object types would be defined — process, data store, source, sink and source/sink. The definitions would specifies that two ‘data store’ objects cannot be linked together, and that ‘source’ objects cannot have incoming links. These rules would be used in providing syntax direction in the editor. Links and objects would in this case carry simple textual content (i.e. data and process descriptions).

6.2.1.3 Data-driven Diagrams

This category of diagrams serve as a graphical presentation of some underlying data. Examples include bar charts, pie charts, scatter plots, connected graphs and the like. Since these diagrams basically display relational information, it is natural to consider these logically as some view (in the database sense) of one or more relations. The use of relations as a basis seems to be fairly adequate and most diagrams in this class can be described.

For diagrams which represent a functional relationship, the independent parameters would correspond to the primary attributes in the underlying relation. Attributes would assume values from either the numeric or symbolic domain. Tuples might be ordered according to numeric primary attributes or symbolic attributes taken from domains like [Jan, Feb, Mar, etc.]. Furthermore, the relation involved would be binary in cases like simple pie charts or graphs, but would be otherwise for, say, superimposed graphs or cumulative bar charts.

In Chapter 5, we have discussed the possible link between the elementary tabular object and an underlying database. The fact that data-oriented diagram objects are also relation-based suggests possible transformation between the two. Dual representation of data in tabular and graphical forms is commonly available in existing packages.

6.2.2 Virtual Editing Functions

For both tree- and network-based diagrams, a set of abstract operations have been identified for maintaining the structure and content of the respective class of diagrams. These would form the basis of editors for specific diagrams. Since data-

oriented diagrams are based on the relational model, we will not repeat here the set of operations for maintaining tabular objects (see §5.2.2.2).

6.2.2.1 Tree Manipulation

The following basic requirements are identified for manipulating tree-based diagrams. In fact, similar facilities have been defined for manipulation of the global document structure (§3.4.1.1).

- A. **Object manipulation.** A specified object could be inserted into a selected slot of an object in a diagram. The loaded object could be newly instantiated (i.e. a template) or a previously saved structure. Similarly, a specified object could be removed.
- B. **Link manipulation.** A non-hierarchical link could be established between two selected objects, provided the syntax definition permits. Existing links could be removed accordingly.
- C. **Data manipulation.** The data associated with a selected object or inter-object link, which is textual in most cases, would be maintained using the appropriate elementary object editor.

The requirements for navigating within a tree-based diagram are basically the same as those provided for equations. In fact, the editor for mathematical expressions described in Chapter 4 can be regarded as an implementation of a specific tree-based 'diagram' editor.

6.2.2.2 Graph Manipulation

The following requirements are identified for manipulating network-based diagrams.

- A. **Object manipulation.** An object of a selected type could be created. Unlike tree-based diagrams, the new object would either be 'dangling' or be attached to an already existent link. An existing object could be removed, in which case all associated links would be removed as well.
- B. **Link manipulation.** An incoming or outgoing link of a specified type could be appended to a selected object, or established between two selected objects, provided the syntax definitions permit. Links could be removed accordingly.
- C. **Data manipulation.** Data associated with objects or links would be maintained using the appropriate elementary object editor.

The following requirements are identified for navigating within a network-based diagram.

- A. **Adjacent objects.** This would highlight all objects which are linked to the one being selected. For directed graphs, incoming and outgoing links should be distinguished.
- B. **Follow path.** This would select an object which is connected to the currently selected object via a specified link. For directed graphs, this would enable either forward or backward trace.
- C. **Arbitrary object.** This would select any specified object arbitrarily.

6.3 ON ADAPTING THE MODEL

We have described a logical model that aims at describing a subset of structure-oriented diagrams. Let us now examine the requirements that would be involved in ‘tailoring’ the model for specific diagrams. Many problems that would arise from an actual implementation will inevitably be neglected in our discussion here. We will focus on four areas, namely, the introduction of specific diagram syntax, the definition of physical constructs, tailoring the user interface, and specifying the layout (or formatting) procedures.

6.3.1 Diagram Syntax

The logical model (or syntactic rules) of a specific type of diagram would be introduced by defining the architecture of all associated types of object and link. These would be the basis for providing syntax direction in the target editor.

As in the document architecture level (§3.4.1.2), there are two possibilities of specifying the diagram syntax. In the graphical approach, the editor (used to specify the syntax of other diagrams) would itself be a tree-based syntax diagram editor. Here, the notation used should be able to describe non-hierarchical relations needed in specifying graph syntax and arbitrary links in a tree.

A requirement of the meta notation is the ability to specify object *instances* as well as types. For instance, from the definition of ‘data link’ for data flow diagrams, one may derive that two objects of the *type* ‘data store’ cannot not be linked together.

```

<data link>    ::=    <process> → <process>
                  |    <process> → <data store>
                  |    <process> ← <data store>
                  |    <process> → <sink>
                  |    <process> ← <source>

```

However, this notation cannot be used to prohibit, say, self-directed links (which is between the same object *instance* rather than two objects of the same *type*). A similar situation would be the prohibition of outgoing links from sharing the same data instance, as exemplified by the requirement that no two transitions from a state in a deterministic finite automata can be due to the same input symbol.

6.3.2 Physical Model

Physical constructs used to compose specific diagrams would have to be specified. At the elementary level, special primitive symbols or icons would be introduced, such as circuit components or the basic compartments in a flow chart. At a higher level, specific composition methods would be introduced. For instance, one would specify that the superscript binding for equations should raise the script box from the base box by a specified amount. These connection methods could be specified declaratively by means of constraint-based approaches introduced earlier. Equipped with these basic composition methods, the formatter would be responsible for determining from the logical structure of a diagram what symbols and connections to use.

6.3.3 User Interface

This process would make an editor look specialized as far as the user is concerned. Although editors for diagrams belonging to the same diagram class are basically performing the same set of virtual operations, different (or specialized) user interaction would be required. For instance, a data flow diagram editor might prompt a user for 'data' which flows along a link just created, while a transition diagram editor might ask for a 'transition symbol' instead, though the two would involve the same internal operation.

The development of user interface management systems (UIMS's) is relevant to the purpose here. These systems provide a set of tools that would automate generation of the user interface for interactive applications. Furthermore, user interaction codes would be isolated from an application program as well as the underlying graphics system. This convenient modifiability of the user interface is basically what we need here.

Based on the current trend, we may well assume the availability of some form of UIMS or similar facilities on most window systems. Literature on the automatic generation of interactive interfaces can be found in [Thomas 1983] [Olsen 1983] [Hudson & King 1986]. The use of constraint-based graphics to generate user interfaces has also been demonstrated [Borning & Duisberg 1986].

6.3.4 Layout Procedures

The automatic layout of diagrams by algorithmic means is common in specialized applications, in which the specific layout problems are well defined. On the other hand, it is doubtful here as to whether the formatting procedures for network-

based diagrams could be generalized and easily specified. Given a circuit diagram's logical connections, it is unlikely that the formatting task (of positioning and routing) can be derived from some 'generalized' procedures which are suitable for other graph-based diagrams too. However, algorithms do exist for deriving the layout for general graph networks [Feiner 1988] [Rowe et al 1987].

In general, life seems to be much easier for hierarchical diagrams since the layout process is routine and requires less intelligence. These diagrams are physically a hierarchy of subpictures which correspond closely to the logical structure itself. Furthermore, at each level the subpictures would be composed according to some well defined way. For instance, one could instruct a flow chart formatter that, to lay out a *while-do* construct, just connect accordingly the decision box and the other sub-picture (which could well be a single procedure box). These routines can easily be specified with a constraint-based tool.

The automatic layout for data-driven diagrams would be trivial for simple cases in which there is a clear correspondence between the attributes in a relation and the parameters of a diagram. One recent work demonstrates use of artificial intelligence to solve the automatic layout problem for complex relational information [Mackinlay 1986]. Given a relation of data, that system would determine the best presentation method by exploring different possibilities (which are in fact numerous). An algebra has been defined to describe the composition of data-presentation graphics, and the selection is based on the measure of 'expressiveness' and 'effectiveness' of a method.

6.4 CONCLUSIONS

In this chapter we study an approach to incorporating a limited set of structure-oriented diagrams in the document model. No implementation has been available to assess its feasibility. Furthermore, some issues are yet to be resolved before the concept can be considered concrete or worthwhile.

Provided user interface management systems (UIMS's) have gained sufficient popularity, the support for automatic generation and customization of user interaction codes should enable us to adapt the user interface of virtual editors for specific diagrams. However, it is still doubtful whether the virtual operations we have defined would definitely suffice for flexible manipulation of different diagrams.

The problem of greatest concern is the adaptability of the formatting procedures. The automatic layout of individual diagram types often requires highly specific knowledge and even algorithms. Different diagram types which may logically be maintained by similar abstract operations may require totally different layout methods. The use of constraint specification to describe geometric layout routines are promising for fairly simple diagrams [Borning & Duisberg 1986], but not for complex graph-oriented drawings. For complex diagrams, it might be necessary for the user to organize the layout manually (which might be more flexible after all!). However, a system that does not provide automatic layout can still be fully aware of the logical structure or syntax of a diagram.

Finally, it must be emphasized that apart from the structured diagram model discussed here, the unstructured counterpart is essential. In fact, conventional drawing facilities (provided by MacDraw and the like) are essential even in a

structure-oriented document system. It can be awkward having to view every drawing from a logical viewpoint (and most drawings are not structured anyway). In some cases, unstructured editors prove to be more efficient and flexible. More discussion of this will be found in Chapter 8.

The description of our document model terminates here. The next chapter will present a critical appraisal of the work achieved in this thesis.

Chapter 7

Assessment of the Unified Architecture

This chapter presents a critical study of the unified approach to formalizing the structural and functional characteristics of document objects, which has been adopted in the definition of our document model. We first revisit briefly the characteristics and assumptions of the structured philosophy. Then follows a detailed evaluation of this approach as implemented in our model. We also try to foresee the limitations of the unified approach when extended beyond our present scope of investigation. Finally, we suggest some future extensions of the work presented in this thesis, both on the document architecture itself and on some possible improvements to the implemented equation editor.

7.1 REVIEW OF THE STRUCTURED FORMALISM

Let us now study again the fundamental principles behind the structured formalism, as outlined in Section 2.1.1, which we have followed in defining the entire document model. A revised version of Figure 2-1 is included here to facilitate our discussion.

- A. **Logical Model.** The structured philosophy requires the object in question to have some logical structure which can be unambiguously formulated by its logical model. In simple cases, the logical syntax may be a program or mathematical structure. However, we will examine carefully in the following sections the implications of "logical structure" for individual objects.

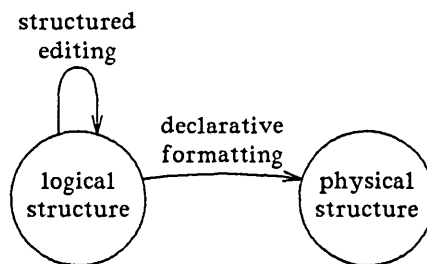


Figure 7-1. The Structured Approach

- B. **Physical Model.** This formalizes the geometrical construction of an object, and typically takes the form of some abstract or descriptive composition methods. Examples include Knuth's box-glue model, the box-binding model for equations, and the grid model for tables. We will later note that a systematic layout model is not necessarily ideal in situations where the physical structure is better described using ordinary graphical primitives.
- C. **Structured Editing.** The structured philosophy requires that a user specifies and manipulates an object in terms of logical entities, assuming these exist, instead of modifying the layout directly. In the following sections, we will examine whether this mode of editing, namely structured editing, is adequate and efficient in various situations.
- D. **Formatting Rules.** The structured formalism assumes that an obvious and clear correspondence exists between the logical and physical constructs defined in the two respective models. These transformation or formatting rules are used in the declarative layout process. Again, we will examine the reality of automated and rule-based layout, including difficult situations like diagrams.

To summarize, the structured formalism requires that some logical and physical forms of an object exist as two distinct structures. Furthermore, it is assumed to be adequate for a user to manipulate only the former (which dominates) via structured editing, while the latter is derived automatically based on some formal mapping procedures. The purpose of this chapter is therefore to investigate the limitations of these principles both within and beyond the scope of our document model.

7.2 EVALUATION OF THE MODEL

This section assesses critically the extent of success with which we have defined the document model by adhering to the structured approach just reviewed. We will examine individually the three elementary objects studied in detail, as well as composite objects defined at the global level. Our discussion will center around four fundamental questions relating to the efficiency of the structured formalism.

- A. Is the logical structure of the object in question naturally and unambiguously defined in the model ?
- B. Does the physical model exist as an abstract mechanism that describes or formulates all possible layouts of the object ?
- C. Is it adequate to manipulate only the logical structure of the object, but not the physical layout, via structured editing ?
- D. Are the logical and physical constructs related via an obvious set of formal rules which enable declarative layout ?

7.2.1 The Equation Object

We first study the performance of the structured formalism when applied to equations. Let us look at the above four issues individually.

7.2.1.1 Mathematical Structure

In Chapter 4, we have assumed that the mathematical structure of an equation is "the logical structure" we need when pursuing the structured approach. We also drew an analogy between the logical view of equations and that of other structured objects like programs. One fundamental difference, however, is that the latter is inherently modeled upon some rigorous and unambiguous grammar. On the other hand, equations merely serve as a convention for human communication, and not for machine interpretation. In Chapter 4, we discussed the problem of ordinary (or mathematically "naive") users having to recognize mathematical structure as the dominant aspect of equations. The subject here, on the other hand, is whether mathematical structure is logical and unambiguous at all, even to mathematicians.

We are asking if a given mathematical expression always yields a unique and unambiguous logical interpretation which can be described as a hierarchy of constructs, as our model requires. Our logical model is based on definitions of constructs having a predetermined number of subordinate elements. The approach is similar in algebraic packages like MacSyma, which models mathematical constructs as parametrized function calls. Let us now examine the limitations of this hierarchical view of mathematical expressions, and explore the possibility that some notations may not be accommodated using this approach.

The example equations considered throughout Chapter 4 all have an obvious mathematical structure, and one would not have any difficulties in identifying their logical structure. The limitations of the hierarchical view, however, may become obvious when one considers some mathematical notations with a comparatively free form. The problem of modeling such unconstrained notations occurs in automated mathematical reasoning, an area which is beyond conventional symbolic computation [Bundy 1983]. For instance, previous work has demonstrated machine solving of mathematical problems in which English text, though from a very limited vocabulary, forms part of the specification [Bundy et al 1979].

Of course, for these comparatively unstructured mathematical conventions, we could have introduced their corresponding production rules in the logical model by "parametrizing" these notations. However, in such ambiguous situations different users are bound to arrive at various solutions when trying to impose a hierarchical view. In the current editor, the users would probably have to define their own templates in such circumstances and, consequently, the editor would not be aware of the intended mathematical structure.

Despite the limitation of the relatively simple hierarchical view of equations adopted in our model (or similar parametric approaches), the scope of expressions is adequate as far as document preparation is concerned. Furthermore, the logical model holds the potential to accommodate a even more complete category of expressions when the set of production rules presented in Appendix B.1 are properly extended.

7.2.1.2 Equation Binding Model

We now move on to the box-binding model we use in building equations. There is no doubt that batch systems like $\text{\TeX}$ are more sophisticated and provide more intelligent formatting decisions. However, we have demonstrated a relatively simple and modest model which achieves presentable composition of equations. Furthermore, we feel that the physical model could be upgraded to achieve quality comparable to $\text{\TeX}$, yet without any modification to the logical model which is defined independently. Although this is not the original objective of our project, we now examine the possibility of improving the existing physical model for better layout quality.

The binding modes we have defined are those that would be needed in rendering most equations. The layout quality does not only depend on the comprehensiveness of the available binding modes, but also on the way these modes are rendered, that is, the second computational step in Section 4.3.4.2. In the current implementation, we have imitated the automatic scaling criterion used by $\text{\TeX}$. However, many other sophisticated formatting procedures in the latter have not been adopted. For instance, we have used a universal horizontal binding mechanism, which means that the amount of padding being inserted between linked components is not adaptable. An ideal formatter should distinguish between a horizontal list making up a function name like "sin" and that composing several complex expressions on the base line, and supply different padding spaces in each case. Furthermore, the present model assumes a uniform type face, and hinders an automatic selection of font as exhibited by $\text{\TeX}$, such as using Roman for function names and italics for variables.

Had the physical model included the appropriate attributes, layout sophistication as exemplified in the above discussion could be achieved via appropriate revisions to the existing formatting rules, as will be discussed in a later section. Of course, this reminds us of the tradeoff between batch and WYSIWYG systems regarding typesetting quality, and there surely exists a limit for interactive systems trying to implement batch layout algorithms. However, the ever increasing computing power might eventually bring this closer to reality.

7.2.1.3 Structured Equation Editing

We now present a thorough analysis of the structured approach to manipulating equations, and try to compare it with alternative methods. In Chapter 4, we concluded that the structured approach to specifying equations is efficient and natural. This "claim" is however fairly subjective. For instance, while we have mentioned that languages like $\text{\TeX}$ are unnecessarily wordy due to extra positioning and delimiting keywords, we have ignored the fact that our structured approach also includes extra actions like mouse clicks (which are extensively needed in template selection, as will be obvious below).

The most appropriate approach to evaluation of the specification methods would be by direct experimental observation of the editor in use. Two issues would be of interest — the learning curve for new users and the behavior of experienced users. Furthermore, a large sample of users must participate in order to yield some meaningful data. Instead, we present here a detailed task analysis of the batch and structured approaches, which serves as a reasonable ground for comparison.

The language chosen to evaluate against the structured approach is $\text{\TeX}$. However, the specification task discussed below should apply to any batch

formatting language in general. We now examine independently the two phases of editing, namely initial specification and subsequent modification.

7.2.1.3.1 Initial Specification

The following discussion is based on an assumption that the target expression has been formulated visually prior to the specification process, either in hand-written or even typeset form.

A. **Batch Language.** Just as there are different approaches to writing a program, a user can arrive at the same T_EX specification in various ways. Here we identify the steps that are generally involved in the specification process.

1. **Identification of horizontal list.** Batch formatting systems express equations in terms of their physical structure, which is reflected in the source specification itself. Independent units appearing sequentially in the input stream are composed horizontally on the current (or local) baseline. The first step in the specification process is typically to isolate the horizontal components in the target equation, and then tackle these individually. For instance, for the expression in Figure 4-12, a user identifies " $\sum_{n=1}^M$ " as the first of the five horizontally linked units, while the summand (i.e. the fraction containing the limit and integral expressions) would be the next to tackle, and so on.
2. **Composing a unit.** A unit identified in the previous step may be broken down further into subordinate units (which surely do not form a horizontal list, or else they would have been isolated in step 1). This non-horizontal unit can be anything from a fraction to a vertical unit (such as $\sum_{n=1}^M$). Since the binding is now not the default horizontal case,

this step requires usage of the correct T_EX reserved words and special format, such as $\{ \dots \} \over \{ \dots \}$ or $\sum_{\dots} \Lambda \{ \dots \}$. The subordinate units, which may themselves be horizontal lists, are then individually analyzed via step 1. Steps 1 and 2 are basically repeated until the entire equation is specified. A T_EX specification for the expression in Figure 4-12 is shown here for reference.

```

\sum_{n=0}^{\infty} \Lambda \{
  {\displaystyle \lim_{x \rightarrow 0} {\sin nx \over x}}
  \over
  {\displaystyle \int_0^{\pi} \cos \Lambda n \theta \, d \theta}
\}
\quad > \quad
\prod_{n=1}^{\infty} \Lambda \{ 2M \} \backslash,
  {\displaystyle n+
    {\strut 1 \over \displaystyle n+
      {\strut 1 \over n+1}}}

```

3. **Trial and debug.** For fairly complex expressions, it is unlikely that an average user can arrive at an error-free specification for the first time. Errors such as mismatched or totally omitted delimiters are common. Apart from the actual specification outlined above, this step of debugging also contributes to the significant latency in the batch approach.

B. Structured Approach. Using the equation editor implemented here, a user would typically follow the steps below when creating an equation.

1. **Target expression analysis.** A user compares the target equation against the current state of the screen, and decides on which subexpression within the former to specify next. This unit must correspond to one of the metaboxes in the equation on screen, although

the sequence in which these boxes are expanded is irrelevant. This step is obviously redundant at the beginning of a session when only one metabox exists.

2. **Cursor adjustment.** After a template selection (to be discussed below), the cursor node would be updated automatically to the metabox which the system "thinks" a user would most likely want to expand next. If the cursor is not currently positioned in accordance with the unit determined in step 1, a user would have to select the desired metabox with a single mouse click. Otherwise, this step is unnecessary.
3. **Template selection.** A user next selects the template library from which a prototype of the expression in step 1 is to be found. After the appropriate menu item is picked, a page of templates is displayed graphically, and the the desired template can be indicated accordingly. This step therefore involves two mouse clicks. However, in the case of matrix templates, one mouse click and a rubber box action are involved instead.
4. **Data specification.** This step is used to specify terminal symbols and identifiers in an equation, and may occur at any stage in the specification process. A user may repeat steps 1 through 3 until the entire structure of the target equation is formed, and then enter terminal data. Alternatively, some data can be specified as soon as their corresponding metaboxes are created, that is, steps 1 through 4 occurring in an interlaced manner.

To specify a single symbol, a user points at the desired metabox while any entry through the keyboard will be effective, that is, no

mouse click is required. However, if a special symbol like Greek letter is required, a user first selects the target metabox, picks the menu item to display a virtual keyboard, and then specify the desired key on the screen. This involves three mouse clicks. Finally, entry of multiple-character identifiers is accomplished with the 'enter text' option, requiring a single mouse click.

At the risk of presenting some biased comments, let us compare the time taken by the author to prepare the equation in Figure 4-12 using both approaches. With the structured editor, he managed to create the expression in about four minutes, while it took him totally about 20 minutes (with several failures) in arriving at an error-free T_EX specification. It must however be noted that the author is extremely familiar with the configuration of menu items and mathematical templates in the structured editor, and has very limited experience with the T_EX language.

Instead of following the procedures outlined earlier for deriving a batch specification, one could alternatively adopt a "structured" approach and expand the specification according to the mathematical structure. For instance, one could imagine selecting a 'greater-than' template by entering "`\quad > \quad`" to start with, and then work on the specifications before and after this line, just like expanding the metaboxes on either side of ">" in our structured editor. In fact, from the author's experience, one tends to make fewer errors by following this structured sequence when deriving a T_EX specification — probably another case in favor of the structured approach.

We conclude that, as far as the initial specification phase is concerned, the structured or declarative approach provides a more natural and efficient dialogue,

especially for mathematically oriented users. The batch approach seems to require a certain degree of expertise from the user before he or she can arrive at a correct specification fluently. This is particularly true for sophisticated languages like T_EX.

7.2.1.3.2 Subsequent Modification

We now examine the modification of a constructed expression using the structured editor and T_EX. For each of these two approaches, we will consider some standard editing tasks.

A. **Batch Language.** The tasks involved in maintaining a language-based specification like T_EX are as follows.

1. **Select.** Selection of a subexpression within an equation is typically achieved by interpreting the linear T_EX specification until the desired unit is identified. Most editors allow a user to mark out an arbitrary range of text. Some may even be customized for handling T_EX syntax, as we have seen in Chapter 2. On the other hand, for primitive editors which do not allow arbitrary selection, a user has to rearrange the specification so that the desired subexpression resides on separate lines by itself. In any case, logical selection is a major problem in the batch approach. The worst situation probably occurs when, say, a matrix column is to be addressed, since matrix specification in T_EX is row-oriented.
2. **Insert.** Provided the new expression is to be inserted simply before or after the selected unit, its specification can simply be entered to the original source accordingly, since horizontal mode is the default binding.

Otherwise, additional keywords and delimiters are needed to specify the relative position of the new expression with respect to the selected unit.

3. **Delete.** This is a fairly straightforward step once the desired expression is selected in step 1.
4. **Move.** Some editors provide this option so that one can move a subexpression about directly using steps 1 and 2. Others may require a user to cut the marked text explicitly to a buffer before retrieving it later. As will be seen below, our editor uses this latter approach instead of providing a direct 'move' option.

B. Structured Approach. We now look at the various tasks of manipulating an already composed equation using the structured editor.

1. **Select.** As documented in Appendix C.2, a user can select any logical subexpression arbitrarily using a rubber box, which is an efficient graphical selection technique. Furthermore, one can switch to the parent or the descendent expression of a currently selected unit conveniently using double mouse clicks.
2. **Insert.** The 'append' functions allow a user to insert a new metabox which can be arbitrarily positioned with respect to the selected unit in eight ways. This task involves two mouse clicks — one for picking the 'append' menu item, and the other to choose the relative position to insert.
3. **Delete.** A selected subexpression can be deleted with a single mouse click to invoke the 'delete' menu item.

4. **Move.** This would involve three steps. First, a user saves the selected unit to the buffer using a single mouse click to invoke the 'cut' menu item. Next, he or she appends a metabox to the desired destination (which necessitates a new selection) using two mouse clicks, as mentioned in the 'insert' action above. Finally, the user pastes the previously saved expression onto the metabox using two mouse clicks — one to select the 'paste' menu item, and one to indicate which buffer to load from. (Three buffers are maintained, as explained in Appendix C.2.) A total of five mouse clicks are required in this task.

The above analysis shows that direct manipulation is in general the preferred approach to modifying an expression. For one thing direct selection of logical subexpressions is more efficient than referring to the textual description in the batch approach. The extreme case is probably illustrated by the need to address a matrix column in a row-dominant formatting language. Furthermore, the menu-driven structured approach eliminates the need to recall any formatter keywords, and also reflects visually any changes to the specification at once.

7.2.1.4 Declarative Equation Layout

The formatting rules presented in Appendix B.3 associate each logical construct with a distinct physical presentation based on the box-binding model. This is the key to automatic or declarative layout of equations, which has been demonstrated in the implemented editor. The knowledge possessed by the editor with regard to automatic layout is basically confined to those constructs which have been implemented as mathematical templates. New logical constructs and their formatting rules are appended to the editor via implementation of additional

templates.

The rules used in the layout process can be modified for different presentation style depending on the context, as depicted in Figure 4-7. Modification of these formatting rules is supported via a set of special functions which, though provided within the editor itself, are not intended for ordinary users. New templates are built into the system using this box editing facility, which allows specification of the precise binding modes and scaling to be effective. However, the actual formatting details, such as the exact drop of a subscript box in number of points, reside in the program codes and cannot be conveniently customized. More discussion of this will be found in Section 7.3.1 when we look at the possibility of decoupling the specification of formatter parameters from the actual codes to an external "style file".

In Section 7.2.1.2, we discussed the possibility of upgrading the box-binding model to include other attributes which are essential for better layout quality. To cope with this more sophisticated physical model, an entirely new and more complex set of formatting rules would be required. For instance, in a multi-font model, the layout rules would specify the appropriate typefaces for function and variable identifiers accordingly. The significance of formatting rules, apart from helping to make layout decisions, is therefore to isolate the mathematical and geometrical properties of equations. We conclude that there is still room for improvement to both the physical model and the formatting rules for higher layout sophistication.

7.2.2 The Tabular Object

We now move on to the second elementary object we have studied in this thesis. Again, we assess the suitability of the structured formalism when applied to tables by examining the four main issues separately.

7.2.2.1 Logical Table Structure

When defining the tabular object in Chapter 5, although we have provided a dual logical representation which aims for both row and column dominance, the structure is basically hierarchical in each dimension. Consequently, only tables with a fairly well structured organization are naturally accommodated in our model. On the other hand, tables with an arbitrary form (such as conference schedules) cannot be described by a simple hierarchy of nested fields. The complex spanning pattern found in these tables would be represented in our model awkwardly by the non-hierarchical spanning links. The resultant "logical" structure would also be closely oriented towards the physical arrangement of table entries. This brings to light the inadequacy of a dominantly hierarchical view of tables which we have taken.

Although we came up with similar conclusions for equations in Section 7.2.1.1, the problem there is to a certain extent tolerable, since the vast majority of mathematical expressions is hierarchical in nature. On the other hand, the restrictive hierarchical view certainly hinders the natural accommodation of some possible and realistic table instances. The logical structure of extremely complex tables is difficult to formulate, and the physical layout is the structure which should dominate in such circumstances. In Section 7.3.2, we will further discuss this possibility of departing from the "logical dominant" philosophy currently

adopted in our model.

7.2.2.2 Physical Array Structure

The grid pattern we have used for composing the physical layout of tables is general enough to describe a diverse class of the object, structured or not. The model is adequate for determining the dimensions of individual compartments (i.e. table entries), thus formulating the global skeleton of a table. However, as we have mentioned in the chapter, some parameters which are essential in realistic table generation have been ignored. These details, though complex and exhaustive, are "auxiliary" in the role of table generation in the sense that they take care of aesthetic issues and presentation flexibility. The purpose of formulating the physical model in the chapter, on the other hand, is to support the task of composing individual compartments to form the required framework for the target table, which is a major step in the formatting process.

The practical issues in table generation are by no means trivial, even when compared to the sophisticated equation layout procedures used by $\text{\TeX}$. The space of presentation flexibility can be illustrated by decisions such as the mode of alignment — both vertical (like decimal adjustment) and horizontal (like baseline adjustment), the style of rulings that partition table entries, and the amount of padding inserted within compartments, just to mention a few. The task of upgrading our physical model to a state which is suitable for realistic table generation would involve the introduction of those missing but essential attributes.

7.2.2.3 Structured Table Manipulation

The structure definition phase for naturally structured tables has been demonstrated in Figure 5-8. This approach outperforms others which require a user to specify procedurally the spanning pattern of field headings (which is already implied in our logical structure for tables). Also, provided the spanning pattern is not complex, it is sufficient for a user to manipulate logical entities alone, such as creating subfields, swapping records, etc.

We mentioned earlier that the logical model fails to accommodate tables with arbitrary spanning patterns in a natural way. Similarly, structure editing alone would not suffice in such situations. The virtual operation discussed in Chapter 5 for spanning entries arbitrarily is a non-structural command which makes up for this inadequacy. A user would create a structured configuration initially, and then use the command to specify whatever spanning requirements in the target table. In the current framework, this "semi-structured" approach is the only means of specifying unstructured table instances, especially schedule types, which have complex alignments.

We conclude that for tables with a highly arbitrary spanning pattern, it might be more appropriate for one to organize directly the physical array structure, that is, to an extent disregarding structured editing. In general, structured manipulation does not always work as satisfactorily for tables as for equations.

7.2.2.4 Automatic Table Generation

We have demonstrated in Chapter 5 the derivation of the grid structure from the logical form of a table, based on a set of tree-to-array transformations. The

declarative formatting principle in the structured formalism has been followed quite satisfactorily. However, we have not demonstrated the workability of the formatting rules for tables with more complex spanning and alignment patterns. As mentioned in the previous section, direct manipulation of the physical layout might in such cases be the only solution, that is, when declarative formatting fails.

Leaving aside the problem of unstructured tables, a user would need more direct control over the layout format. This is evident from the degree of control provided in TBL, which is also intended to be a utility for the end-user. This again raises doubts about the practicability of the simple "rule-based" model of the layout process, which seems to drift even more from reality when we discuss diagrams in the next section.

7.2.3 Structured Diagrams

We now study the third elementary object presented in this thesis — diagrams. We have taken a totally different approach when we tackled diagrams. Instead of presenting a successful formulation of the object as part of the entire document model, we have only discussed in Chapter 6 the possibility of imposing the structured view on diagrams, and in particular that of generalizing the requirements for modeling multiple classes of diagrams. Here, we will first take a critical look at the motivation itself towards a generalized model, and then discuss the four issues previously examined for equations and tables.

7.2.3.1 Motivation Revisited

We have claimed in Chapter 6 that despite the diversity involved, it is worthwhile including diagram in its generalized form in our model. The reason put forward was that a structure-oriented document system should apply its structured philosophy to virtually all objects. Whether this generalized approach could be feasible at all is another question, which will be examined in the following sections. Here, we raise two fundamental issues concerning the motivation itself.

- A. **Scope of diagrams.** The scope of diagrams which could be described using a structure-oriented view is fairly narrow. For instance, the number of diagrams we have quoted in Chapter 6 as examples fails to justify development of a generalized model. If only a handful of diagrams could be accommodated, it might well be more appropriate to model these as individual objects, since the value of a customizable model would be questionable.
- B. **Proportion of sharable codes.** The second consideration is the practical benefits of customization, namely the amount of codes that could be saved. Significant resemblance among various diagram editors and formatters is essential to imply minimal efforts in the customization process. The actual amount of sharable codes depends on the editing and layout requirements for various diagrams. The idea of a generalized approach is therefore based on the assumption that diagrams from the same class share significantly similar editing and layout requirements.

7.2.3.2 Logically oriented Diagrams

Leaving aside the question of motivation, we now examine the feasibility of a generalized model by first discussing in this section the meaning of "logical structure" for diagrams, and whether our model manages to specify it in a generalized manner. The implication was fairly vague when we used the term "logically oriented diagrams" in introducing the elementary object 'structured diagram' in Chapter 3. However, after reading Chapter 6, one might realize that we are effectively referring to scientifically oriented diagrams.

By and large the "logical structure" we have been referring to in the chapter is some specific (and rather abstract) meaning one would perceive from a perspective which is relevant to the purpose of the diagram. This contrasts with physical structure which reflects the geometrical connections one may only perceive visually. Of course, the physical layout of a diagram undoubtedly provides an essential clue to identifying its logical structure, hence our later discussion on whether it is always adequate for one to be only able to interact in logical terms (i.e. purely structure editing).

Most diagrams possess some unambiguous logical structure one way or the other. However, the question is whether it would be feasible at all to include their definitions in a generalized model. Diagrams such as architect's plans or chemical formulas are only examples of many diagrams having some unique logical implications, but which can hardly be generalized along with others. The idea of organizing all possible diagrams into a finite number of categories (namely illustration classes) is therefore not sound. The three diagram classes provided in Chapter 6 only represent a narrow domain which is inadequate.

Furthermore, diagrams which are categorized under the same illustration class only share an "identical" logical structure at a global level, but still exhibit dramatic differences, and deviate from the prototype model. For instance, although the topological configuration of all network-oriented diagrams may be described in the generic definition of that diagram class, the actual attributes and data types required in formulating the logical structure of individual diagrams still vary significantly. The generic definitions presented in Chapter 6 for the three diagram classes seem to be inadequate for formulating the logical structure of any particular diagrams.

7.2.3.3 Physical Diagram Structure

Graphical applications in general do have an abstract mechanism for describing the physical composition of objects they handle, that is, geometric modeling. The structured approach being pursued in this thesis, however, requires that such model be specified formally in order to facilitate the subsequent specification of the layout process in terms of some formatting rules. This restriction certainly imposes a serious problem, since the layout of diagrams cannot be always formalized in this manner. In fact, only those objects which are to an extent physically constrained, such as equations, lend themselves to an obvious formalization in the layout model. The difficulty in formulating the physical layout of diagrams, and hence the formatting process itself, is one of the major factors that lead to doubts whether the structured approach is feasible for diagrams at all. In fact, this to a certain extent gives us some good hints on the possible limitations of the structured formalism itself.

7.2.3.4 Structured Diagram Manipulation

Assuming the diagram in question has an unambiguous logical structure, we here discuss whether it is adequate for a user to manipulate a diagram only in terms of its logical entities. It may not be always desirable that one could only specify in terms of logical but non-graphical contents. This form of editing only works satisfactorily for diagrams with a highly rigid and predictable layout, where the user's concern is basically for the logical content only. This is best illustrated by data-presentation diagrams for which the user basically needs to maintain the underlying data only. Structured editing certainly works well in such cases. Due to the very nature of diagrams, however, a user often needs to have more direct control over the actual layout. The inadequacy of structured editing, enhanced by the difficulty in declarative layout discussed next, poses as another setback in adhering to the structured approach for diagrams.

Furthermore, the generalized abstract operations defined in Section 6.2.2 for maintaining the generic models would not be adequate for specific diagrams. Only actual implementation could assess the inadequacy that would arise for individual types of diagrams. This is an issue concerning the motivation of having a generalized model, rather than the structured formalism itself.

7.2.3.5 Automatic Diagram Layout

The structured approach implies that there exists an obvious mapping between the logical and physical constructs of a diagram, so that the latter can be derived automatically from the former. The formatting rules used in the equation editor may help to demonstrate this point, if one treats equations as hierarchically nested "diagrams". However, the automatic layout of equations is comparatively simple,

since, given the mathematical structure of an expression, there only exist finite ways of physical presentation. This is true even if one takes into account the possibility of alternative formatting rules for some constructs.

The situation is different in the layout of real diagrams. Although in Chapter 6 we concluded that automatic layout of tree-based diagrams seemed feasible (in contrast to the uncertainty for graph-oriented diagrams), doubts about its reality do exist. Take as an example the simple case of an organization charts. The internal hierarchical structure of such diagrams may be unambiguous. Yet, the layout which declarative formatting would probably generate is at best acceptable, but not necessarily always the form intended, due to the infinite possibilities of spatial arrangement among diagram elements. Layout of graph-oriented diagrams without any user intervention seems to be an even more remote possibility, especially when these procedures are to be derived from a generalized set of transformation rules.

This reveals the limitations of automatic layout, and brings out another important point with regard to specialized drafting tools we discussed in Chapter 6. Specialized systems (like CAD tools) do not necessarily follow the structured formalism, at least when declarative formatting is concerned. A user could be responsible and be able to position diagram elements arbitrarily in the preferred layout. The intelligent system would then be able to derive the logical structure from the geometric connections, which is basically the 'analysis' step in Figure 2-1. This seems to be a more plausible approach than automatic formatting, which either produces undesirable layout or even fails totally within the structured formalism.

7.2.4 Composite Objects

After studying the three elementary objects, let us now examine how the structured philosophy performs for composite objects at a more global level in the document hierarchy.

7.2.4.1 Global Document Structure

Based on the scope we set out in Chapter 3, the class of documents we are interested in are those having a dominant and obvious logical structure. This to some extent narrows our definition of "documents" to a fairly incomplete category. The example document we examined in Figure 3-4 when discussing the scenario of specialized editors being dispatched in a stacked manner, for instance, is only a highly structured instance which is not representative in a strict sense. There do exist documents which have no obvious logical structure. These layout-oriented materials are not uncommon, and cannot be accommodated naturally in our model.

Despite this fact, the global logical structure of a document in general can be identified unambiguously. Although the definitions presented in Appendix A may not be comprehensive, this point can be demonstrated by the environment libraries in working systems like Scribe [Reid 1980a] or L^AT_EX [Lamport 1986]. The composite objects provided in these systems are adequate for organizing logically a fairly diverse class of documents.

7.2.4.2 Page Layout Model

The page model presented in Chapter 3 is not adequate to support a working composition system. That is basically due to the limited scope of the investigation

set out in Chapter 2. Let us now examine whether the page layout structure can be formulated using a uniform model at all, as required in the structured approach.

There is little doubt that standard layout options associated with individual composite objects (such as leading, margin extent, point size, etc.) can be parametrized accordingly in the page model. On the other hand, global layout characteristics do not lend themselves to formalization as naturally as the above standard parameters. These are basically the complex constraints among composite objects, such as the order of nesting or overlapping among composite areas, continuation of flowing content from one area to another, and the like. Nevertheless, formal definitions of the page layout model have been demonstrated in the ODA standard [ECMA 1985].

7.2.4.3 Structured Document Manipulation

Early structured editors are confined to handling computer programs. Later systems extended structured manipulation to ordinary documents, although the constructs being identified are generally limited to sections, paragraphs and the like. Current text editors have demonstrated the power of structured editing through so-called outline processing functions. Apart from pure structure manipulation, however, a user may occasionally need to specify directly the page layout in the preferred style. Problems do exist in these circumstances if the formatting styles are rigidly defined and shielded from the user, as in Scribe. Furthermore, when a system assumes a particular architecture or composition for the composite objects, structured editing in the strict sense could be too restrictive. It might be more appropriate to confine this form of manipulation to constructs with an obvious structure (like paragraphs and sections in the simplest case).

7.2.4.4 Page-formatting Rules

In Chapter 3, we have demonstrated the use of formatting rules in determining the presentation styles for composite objects. However, it is doubtful whether the entire page composition process can in fact be entirely rule-driven. The page layout problem would be overly simplified if one views it as a mapping from the composite constructs to the physical counterparts. Nevertheless, in the ODA standard introduced in Chapter 2, the formatting process is formally represented with a set of 'correspondence rules' in this manner. However, the standard merely presents a notational description of the activity, and does not imply that the formalism has served any actual implementation.

7.2.5 Summary of Appraisal

Let us now summarize our evaluation of the structured philosophy when applied to various objects in our model. The hierarchical view we have taken of equations is adequate in most situations, except for rare but possible mathematical notations. The box-binding model is adequate for our purpose of investigation, but lacks a comprehensive set of attributes needed for more sophisticated layout. Rule-based automatic layout and structured manipulation have been demonstrated satisfactorily. On the whole, the structured approach works very well for equations.

On the other hand, only structurally organized tables are accommodated satisfactorily in our model. Tables with complex spanning patterns are awkwardly represented in the model, and are not adequately maintained using structured operations alone. Because tables are less constrained structurally than equations, the logical formalism works less well.

Finally, our work on diagrams using the same philosophy does not yield any promising results at all. Difficulties occur in all the four aspects of the structured formalism. However, our attempt to impose the structured view on diagrams is worthwhile. In fact, the difficulties we have encountered with this object give us considerable insight into the limitations of the approach.

7.2.6 Limitations of the Unified Framework

We have examined in detail the shortcomings of the structured approach to document modeling within the context of objects actually defined in our architecture. We now take a broader view and try to foresee the performance of the structured philosophy when applied beyond the scope of investigation in this thesis. In other words, we are here examining the reality of the unified philosophy, that is, having a homogeneous model for a diverse class of objects.

The philosophy of some form of unified model has appeared in data modeling formalisms other than document structuring. Similar goals have been applied to, say, formulating office objects or even office tasks. However, we will here discuss the approach within the context of this thesis, namely document architecture, since the model we have built supports document formatting systems.

In Chapter 3, elementary objects were defined to be those which possess some unique structural and functional properties, lending themselves to the structured formalism. The assumption that a generic and homogeneous model could be applied in exactly the same manner to individual elementary objects has surely limited the scope of documents that could be accommodated. Even for those objects which the model has included at present, there are still some difficulties one way or the other, as seen in the previous section. This limitation is even more

obvious when one takes into account existing work on multi-media document architectures, where the nature of objects of interest often deviates drastically from our assumptions reviewed in Section 7.1.

When we evaluated our model in the previous section, it was revealed that individual objects may perform satisfactorily in some aspects of the structured formalism but fail in others. Nevertheless, one can easily generalize the situation, and foresee the problems with the structured formalism if we try to extend the scope of the existing model. The following comments basically reflect the difficulties that concern one elementary object or the other in our model.

- A. **Nonexistence of logical structure.** The first problem is obviously a possible ambiguity or even failure in identifying the logical structure of the object in question. In many circumstances, the geometric structure is the only obvious form that an object assumes. The 'equation' object is by far the one with the most unambiguous logical structure. For diagrams, we have already seen that only a small set of special cases possesses some well-defined logical structure. The definition of "logical" structure will certainly be even more vague when it comes to objects from other unconventional media.
- B. **Informal physical model.** Another difficulty is in formulating the geometric layout formally, which in most cases is best represented using basic graphical primitives alone. The failure of arriving at a descriptive specification of the physical model may necessitate a deviation from the "rule-based" formatting practice. As will be seen later, this fact does not necessarily hinder automatic layout itself, but only calls for revisions to the existing model of the formatting process. Again, the definition of physical structure could be just as vague as that of its logical counterpart for objects in a multi-media

situation.

- C. **When structured editing fails.** There would be no point of discussing this issue if the logical model for an object cannot be formulated in the first place. Even though the logical structure is successfully identified, structured editing is not necessarily an efficient solution. This fact is best illustrated by the example of program editors, as programmers do not necessarily develop their codes in the expected structured manner. Our equation editor also reveals some of the shortfalls of structured editing, as discussed previously, although on the whole its performance is promising. In general, structured editing could be inefficient or even inadequate under some circumstances, and its performance should be assessed individually for different objects.
- D. **Difficulties in rule-based layout.** Here one can basically arrive at two situations. In the first case, this activity would fail once the logical and/or physical models cannot be formulated successfully. The other possibility is that although the two structures are reasonably well defined, an association between the two cannot be established via a set of unambiguous rules which are essential in the formatting process (as far as the structured formalism is concerned). However, it should be noted that specialized formatters need not follow the restrictive rule-driven approach implied by the structured philosophy.

The structured formalism has obvious limitations. Although the 'equation' object in our model has displayed satisfactory performance using this approach, the same cannot be applied to virtually all other objects without compromise. It should not be assumed that *all* objects have a "logical" and a "physical" structure, that it is necessarily more efficient for a user to manipulate the former, and that the layout

should always be derived from the logical form which dominates.

Consequently, the idea of unified model based on a homogeneous structured approach must not be carried too far. We must admit here that there is not much room for our model to extend before reaching its limit. Hence our discussion of future work towards a "hybrid" model in Section 7.3.2. Furthermore, the restrictive structural view does not seem to fit naturally into existing multi-media document architectures, where the definition of logical and physical structure is too vague. Our appraisal of the formulated model ends here, the next section will be a proposal of future work.

7.3 FUTURE DIRECTIONS

After reviewing the shortfalls and anticipating the limit of our model, let us now examine some areas where there is still room for further research or implementation. We first look at several issues concerning the equation editor which deserve further investigation. Next, we propose a revised document modeling approach which incorporates various changes to the formalism currently adopted.

7.3.1 Revising the Equation Editor

Several major and plausible improvements to the implemented editor are possible, along with some subsidiary issues. These are discussed below in an increasing order of perceived complexity.

7.3.1.1 Alternative Specification Methods

In the current editor, all functions are invoked via mouse clicks and menus, and this results in awkward circumstances. A typical situation is the specification of expressions like $x^2 + y^2 = z^2$. The superscripts in this expression have ruled out the possibility of direct input using the 'enter text' option. Suppose the 'equality' and 'sum' templates have been invoked to produce $\square + \square = \square$. Specification of each of the three boxes alone would involve a selection of the 'superscript' template (which requires two mouse clicks), and positioning the pointer properly before entering the actual symbols. A desirable improvement would be to provide alternative commands mapped to the keyboard input sequence, so that the above expression could simply be entered in a single stream, say $x\wedge 2 + y\wedge 2 = z\wedge 2$.

One of the main factors that hinders efficient specification in the current editor is a strong reliance on the use of templates, even for structurally simple expressions. (It must however be noted that the use of templates is a strength of the editor when complex expressions are concerned.) A plausible improvement would be to fully support non-template-driven alternatives in the specification process, but yet include a simple parsing step to reconstruct the mathematical structure when necessary. For instance, after entering the above expression as an ordinary text stream, a user should still be able to, say, select the expression on the left of the equality as a logical unit. The issue of recovering the mathematical structure will be discussed later.

7.3.1.2 Automatic Symbolic Substitution

The existing editor has no knowledge whatsoever as to which symbols within an equation are logically related. This has hindered a common and useful editing function, namely the automatic substitution of variable names. Certainly, in the case of equations, a blind global substitution is not always intended, since a symbol name can be reused in different context, even within a single equation.

There are three possibilities for implementing this function. The first method requires a bit of intelligence in the editor. We could include a set of "scope rules" in the logical model which governs the reuse of symbol names (as in a programming language). The system would then be able to decide that, for instance, identical symbols appearing separately in the 'numerator' and 'denominator' fields of a 'fraction' construct are logically distinct, while those appearing in the 'index' and 'summand' fields of a 'summation' object are logically related. This approach, as in any algebraic packages, would allow an implicit

declaration of the semantic structure of expressions.

The second approach would require a user, upon creation of an expression, to tag explicitly those symbols that are logically identical. This could be an awkward process, as one can imagine. The third approach, which compromises between the need for system intelligence and a user's effort, would be to require that the user specifies the extent (using the rubber box cursor) within which a global substitution can be carried out. It seems that in this case a user might be in better control than in the first approach, where the intelligent system would tend to dominate in the symbol substitution process.

7.3.1.3 Layout Control Mechanism

As mentioned before, the existing binding model has inevitably overlooked some fine points in equation setting. A plausible solution would be to include some user control in the layout process. In the simplest case, invisible boxes of various dimensions could be provided for a user to insert anywhere within an equations in order to make up for the formatter's imperfection.

In the current editor, formatting rules are specified by actually composing boxes and arranging them graphically (using the 'append' function). This could be quite a laborious process, and subsequent maintenance could be awkward as well. Furthermore, if the physical model were to be upgraded for more presentable layout, as suggested in Section 7.2.1.2, the formatting rules are bound to be more complex. It may seem worthwhile to investigate the possibility of representing these comprehensive rules in some editable and expressive format which is interpreted by the formatter. Furthermore, the actual layout parameters could then be decoupled from the formatter itself and reside in these external "style

libraries". A higher degree of customization should then be possible.

7.3.1.4 Postprocessing by Batch Formatter

Due to the inherent discrepancy between the available resolution on display and print devices (at least for the present moment), what one sees on the screen can never be exactly what he gets on the paper. This effectively imposes an upper bound on the performance of interactive formatters, as far as layout quality is concerned. It follows that the purpose of refining our physical model, as discussed earlier, is entirely for the sake of ultimate print quality, rather than for display improvements. Since the screen layout is not going to benefit substantially from an upgraded physical model (which will certainly introduce a high degree of complexity to the system), it may be worthwhile to consider alternative ways of improving the output quality.

One approach commonly adopted in interactive systems is to compile the output in an existing formatting language. One possible extension to our editor, therefore, is to generate its output in $\text{\TeX}$ format. Furthermore, apart from this motivation, some journals actually recommend $\text{\TeX}$ as the preferred electronic format for manuscripts. Mapping from our box-binding model to $\text{\TeX}$'s constructs seems to be a feasible task, as the latter are more expressive than is required to describe the relatively primitive box-binding constructs. However, among other unforeseen problems, one difficulty would be the decision on which of the countless $\text{\TeX}$ parameters should be accessible to a user. The degree of control that can be provided in a non-language interface certainly has its limit.

7.3.1.5 Interface for Symbolic Packages

Although symbolic packages like MacSyma present expressions neatly and instantly on ordinary terminals, specification is based on a parametrized language format and, more importantly, manipulation of expressions can be tedious. Furthermore, the lack of typesetting support in these packages often hinders a desirable integration of computation and documentation tasks. This accounts for previous efforts on graphical front-ends to existing algebraic packages. As mentioned in the introduction chapter, the syntactic knowledge possessed by a document system not only benefits the editing process itself, but also opens up the possibility of collaboration with external tools.

Due to the resemblance of our model and that of, say MacSyma (in terms of the latter's hierarchical view, not its comprehensiveness, of course), it seems possible that the editor could serve as an interactive front-end to the latter. One possible scenario is that a user composes and modifies his problems interactively using our editor, sends these across to MacSyma at a remote site, and then views the returned result graphically on the editor. This is the philosophy behind another system (CaminoReal) reviewed in Chapter 4 [Arnon et al 1988].

The first implementation issue involved is the compatibility between the mathematical constructs available in MacSyma and our editor. Although our present model only contains a subset of the objects provided in MacSyma, we cannot foresee any major technical difficulty in extending the model. The second consideration concerns our editor's compilation of MacSyma source, and the interpretation of results returned by the latter. The MacSyma syntax is structurally fairly close to external storage format used in our editor, except that we use a binary instead of a human-readable format. Although the actual

interface with the symbolic package could involve many unforeseen difficulties, it appears to be a plausible direction which is worth pursuing.

7.3.1.6 Reconstruction of Mathematical Structure

In the current editor, the mathematical structure of an expression is maintained only if templates have been used consistently throughout the specification process. On the other hand, if one were to specify the geometric composition of any part of an expression directly using the 'append' option, the full mathematical structure would not be captured by the system. Furthermore, the 'append' mode is sometimes inevitable when one needs to modify an existing expression (unless it is rebuilt entirely from scratch using templates). A similar situation occurs if one uses the 'enter text' command to specify character streams other than user-defined variable identifiers, such as " $x+y=0$ ".

The mathematical structure of an expression is not essential for most operations in the editor. However, the above problems must be tackled within the context of an earlier proposal to extend the editor into an intelligent interface for algebraic packages like MacSyma. One plausible direction, therefore, is to investigate the reconstruction of the mathematical structure of an expression from its geometrical representation.

Recovery of the mathematical structure from linear text streams (as exemplified above by " $x+y=0$ ") would involve a simple parsing scheme, and we do not anticipate much difficulties in implementing this function. On the other hand, identification of geometrically complex constructs would require a deeper investigation. Although the problem is basically an inverse of the formatting process, it will not be a trivial one, and would involve a higher degree of

intelligence than in the forward step of declarative layout. The problem is even more complicated if the logical structure were to be rebuilt "incrementally" while a user is composing the layout, since at certain stages the latter may not correspond to any valid mathematical form at all. We feel that this problem is worthwhile for future investigation.

7.3.2 An Integrated but Hybrid Framework

We now look at a possible approach to compromising the restrictive structural view adopted in the unified architecture. We now present an overall picture of the proposed approach, highlighting the fundamental differences from the current architecture, and then study individually the issues that need further research.

7.3.2.1 An Overview

We have already seen that the adherence to the structured approach poses problems within and beyond the scope of our thesis. It seems worthwhile to investigate the possibility of some form of compromise solution as an alternative to the unified, "one model for all" approach. Here we propose that future extension of this project should aim for an integrated but hybrid model, in which the structured philosophy is not rigidly pursued. A model which is less restricted by this formalism should accommodate more objects, more efficiently. In the following discussion we will not indicate the specific scope of objects that the new model should accommodate. Instead, we only focus on the philosophical changes to the unified approach which are essential in overcoming its limitations.

However, compromising the structured formalism does not imply our abandoning the goal of integration, which is the purpose of defining a document architecture in the first place. We feel that the idea of having specialized editors and formatters for individual objects working in a stacked manner is still sound, but that their internal operation should be "adaptive", rather than being restricted to the formalism.

In Chapter 2 we have reviewed some current researches on achieving a compromise between direct-manipulation and language-based approach to the specification process. The former is generally accepted as the favorable choice but does prove less efficient in certain situations. For instance, although sketching programs like MacDraw are in general accepted as an efficient and handy tool, their language-oriented counterparts definitely do the job better when a high degree of precision and "programmability" is required. It has been shown that each of the two approaches performs better for some objects and during certain steps in the specification process. The philosophy behind current work in this subject is that a compromise should be sought instead of adhering to either solution.

Although the issue here is not one concerning interactive versus language-based specifications, the same argument applies equally well. Instead of imposing the same formalism to all objects at all levels, future investigations should be directed towards a model which allows totally different frameworks to exist among various objects. The suitability of the structural formalism should be assessed individually for different objects.

For some objects in the extreme case, the scenario set out in Section 7.1 might be totally abandoned. More often, however, we would expect that one constraint or another need to be relaxed, or even some alternative activities be augmented to the

model. A more concrete example will be illustrated by a later discussion about extending the equation editor through the provision of two additional activities, namely, layout editing and geometrical-to-mathematical transformation.

7.3.2.2 Non-dominating Logical Structure

It has been assumed throughout the discussion of our model that the physical layout is always a product derived from the logical counterpart. Another assumption (that a user would naturally maintain the latter structure alone) further strengthens the implication that the logical structure is the dominating form of an object. In reality, as we have seen, the layout structure may be the natural or even the only form some objects may assume. Furthermore, manipulation of the logical structure alone is inadequate in some situations.

In the proposed architecture, one should look into the possibility of the two structures changing role for some objects. The logical form might become the derived structure instead, as exemplified by our earlier discussion of reconstructing the mathematical structure of an expression from its typographical layout. The new model should also allow the total nonexistence of the logical structure, since it is the physical layout that is essential, as far as document preparation is concerned.

One could even look into the possibility of having a mixed representation for both logical and physical constructs within a single structure, and study the problem of local transformation between the two. Again, the equation object serves a perfect scenario. As we have seen, users of the editor may not exploit templates consistently, resulting in an incomplete mathematical structure at some stages (that is, logical and geometrical constructs coexisting in the same structure).

Future investigations should therefore include a study of a mixed architecture which contains arbitrarily both logical or physical constructs, and the transformation between the two freely, without assuming any dominant form. This could be a suitable alternative to either logical-dominant or physical-dominant representations. However, the optimal representation depends upon the object in question. The compromise required for individual objects should also be part of this investigation.

7.3.2.3 Multiple Modes of Editing

The symmetric roles played by the logical and physical structures as indicated above affect the roles of the user and the system as well. Instead of providing a single set of structure editing functions, an alternative activity for maintaining the layout should be specified in the model too. For instance, apart from the template-based operations, a specialized equation editor should also include special box-editing functions which are based on the geometry of equations. (Box manipulation is not conveniently supported in the current editor). Similarly, a table editor should include functions which arrange the grid pattern and alignment among table compartments, apart from the functions described in our abstract editor.

In general, structure editing should not be the only activity defined in the processing model. Future work should investigate the functional integration of structure and layout editors working on the same hybrid structure proposed in the previous section.

7.3.2.4 Alternative Formatting Mechanism

One of the major limiting factors in the existing model is the assumed mechanism in the formatting process. Future work should explore alternative representations which are not confined to the rule-based mapping model. The current view that the process is based on a simple transformation mechanism is too narrow indeed. The layout process in the equation editor is not entirely defined in the formatting rules, since the second computational phase has not been specified formally. Since the layout procedure is an implementation issue, one may be doubtful whether the formatting activity need to be formalized at all.

This question actually reduces to the purpose of defining the document model in the first place. If one accepts that the document architecture serves to achieve object integration and a uniform editing interface, a formal definition of the internal mechanism of the formatter is not essential at all. On the other hand, if the architecture aims to serve as a full specification of an abstract system, as the current model intends to, future work should aim for a more practical and alternative approach.

7.4 CONCLUSIONS

In this chapter we have presented a critical report on the degree of success with which the structured formalism has been pursued at various levels of our document model. The investigation reviews that the structured approach works well for equations, which have a well defined and inherent logical structure. On the other hand, this philosophy is not ideally suited to tables, which have an relatively unconstrained structure. The rigid formalism is not that promising at

all when it comes to diagrams.

Despite the limitations of a unified architecture adhering to the structured philosophy, the latter is definitely desirable within its scope of applicability, as exemplified by the equation object. Furthermore, the unsuccessful attempt to structuralize diagrams based on an identical approach exposes the difficulties of having a unified framework for diverse objects. This leads to our discussion of possible future directions towards a universal framework which does not insist on the structured philosophy. We have also identified some plausible improvements to the implemented equation editor that should be investigated.

In conclusion, the work presented in this thesis has demonstrated the power and exposed the limitations of the structured philosophy by exercising it to an appropriate but not excessive extent.

Chapter 8

Conclusions

In this concluding chapter, we will first summarize the work presented in this thesis. Next, we will discuss the prospects of structured and integrated document systems, noting the advantages offered by their unstructured counterparts. One should refer to the previous chapter for a detailed assessment of the structured formalism as pursued in our document model, as well as some proposed future directions for this research.

8.1 SUMMARY OF ACHIEVEMENTS

The achievements of the work presented here can be summarized as follows.

A. The design of a structured document model.

A logical document model has been defined which aims at overcoming the inadequacy of existing architectures. Emphasis has been put on a more comprehensive structuring of elementary document objects. Three objects that commonly occur in scientific documents (namely, equations, tables, and structured diagrams) are studied in detail. The model serves to demonstrate the architectural approach to designing an integrated and structured system. An abstract design has been presented in which a crew of specialized object editors would be dispatched by a global document structure editor.

B. The implementation of a WYSIWYG equation editor.

Based on definitions of the object *equation* as part of the document

architecture, an interactive equation editor has been implemented as an illustration of a specialized editor. This is selected due to its practical value and because existing structured equation editors lack a comprehensive logical view of mathematical expressions. The editor's knowledge of mathematical syntax is incorporated in the form of *mathematical templates*. The template-driven and structured approach to manipulate equations has overcome some disadvantages of similar tools. Furthermore, the geometric and syntactic information embedded in the templates has enabled a *declarative* approach to describing mathematical expressions, which has previously appeared in document specification.

C. The proposal of a generalized model for structured diagrams.

A generalized model has been proposed to describe a limited set of logically oriented diagrams. The motivation is that a structured document system should provide diagram-specific syntax direction, that is, being specialized only at the syntactic level. Due to the diversity involved, three *illustration classes* have been defined for diagrams having a similar underlying structure. Diagrams belonging to a certain class would share an *abstract model* and a corresponding *virtual editor* accordingly. No implementation has been available to assess the problems that would be involved in adapting the model for specific diagrams, although these have been studied.

D. The proposal of a statistical font caching scheme.

A caching scheme that exploits the statistical nature of English text has been proposed for interactive environments for displaying high-resolution text. The scheme would be particularly useful where there is a necessary trade-off between storage space and generation speed. Simulations have demonstrated better performance than memory management schemes that do not exploit the

statistical properties of English. The best results were obtained by replacing the least frequently used character whose probability, predicted with trigram statistics, falls below some threshold.

8.2 STRUCTURE versus FLEXIBILITY

The benefits of a structured approach to document manipulation are perhaps best illustrated by the example of equation editing discussed in Chapter 4. The knowledge of mathematical expressions which both the system and the user possess has resulted in an efficient and natural dialogue. These benefits are further highlighted by the trend that authors are themselves increasingly involved in document preparation work. However, the strength of that example application also implies two fundamental limitations of the structured approach.

First, the object in question must have an inherent logical structure. Many objects are, however, not structured naturally. A system which carries the structured philosophy too far might insist on logical organization of all objects it handles. This would either result in many unstructured but necessary constructs being totally excluded in the model, or some objects being organized in an awkward way.

The second limitation of the structured approach is that even with logically oriented objects, a user must have the necessary knowledge to communicate with the the system efficiently. While using the structured equation editor, for instance, a user without the necessary mathematical knowledge would probably have to rely totally on the *append* options. As another example, an ordinary user might not be able to draw a flow chart at all using its specialized editor, but could do so

comfortably with a conventional graphics editor like MacDraw. A structured but flexible document system should therefore preserve some procedural means of manipulating document objects which do not necessitate any specific knowledge.

These two weaknesses of structure oriented systems have reflected the strength of their unstructured counterparts — flexibility. For instance, an ordinary WYSIWYG editor would offer document templates for convenience but would still allow a document to deviate from this predefined format. On the other hand, a strictly structured editor might require a user to create documents according to definitions in the architecture.

In the abstract system presented in Chapter 3, it has been proposed that object integration should be extended to the functional level, that is, specialized editors for various objects working in an integrated manner. This again remind us of the fact that existing composition applications are remarkably flexible in integrating objects which can be prepared using a diverse class of packages. While the architectural approach emphasizes integration and structuring of specialized applications, flexibility is definitely sacrificed.

As a final remark, despite the inflexibility we have just addressed, the structured approach to document manipulation is definitely more efficient as a disciplined document preparation environment where structured objects are involved.

REFERENCES

- [Allen 1981] T. Allen, PEN : a Hierarchical Document Editor, *ACM Notices on Programming Language* 16, 6 (June 1981), 74-81.
- [Angell et al 1986] I. O. Angell, Y. P. Low and A. R. Warman, Genie-M : a Generator for Multimedia Information Environments, in *Proceedings of the International Conference on Electronic Publishing*, Wembly, UK, November 1985.
- [Arnon 1987] D. Arnon, Report on Workshop on Environments for Computational Mathematics (during ACM SIGGRAPH Conference), Internal memo, Xerox Palo Alto Research Center, July 1987.
- [Arnon et al 1988] D. Arnon, R. Beach, K. McIsaac and C. Waldspurger, CaminoReal : an Interactive Mathematical Notebook, in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography*, Nice, France, April 1988.
- [Ayers et al 1984] R. M. Ayers, J. T. Horning, B. W. Lampson and J. G. Mitchell, Interscript : a proposal for a Standard for the Interchange of Editable Documents, Technical Report, Xerox Palo Alto Research Center, 1984.
- [Bair 1987] Bair, Report on Computer Supported Cooperative Work '86, *ACM Notices on Office Information System* 8, 3 (1987), 3-13.
- [Beach & Stone 1983] R. Beach and M. Stone, Graphical Style towards High Quality Illustrations, *ACM Computer Graphics* 17, 3 (1983), 127-135.
- [Beach 1986] R. J. Beach, Tabular Typography, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 18-33.
- [Beatty et al 1979] J. C. Beatty, J. S. Chin and H. F. Moll, An Interactive Documentation System, *ACM Computer Graphics* 13, 2

- (August 1979), 71-82.
- [Bentley & Kernighan 1984] J. L. Bentley and B. W. Kernighan, GRAP - a Language for Typesetting Graphs, Computing Science Technical Report No.114, AT & T Bell Laboratories, December 1984.
- [Beretta et al 1982] G. Beretta et al, XS-1 : an Integrated Interactive System, *Proceedings of the 6th International Conference on Software Engineering*, Tokyo, Japan, 1982, 340-349.
- [Berns 1968] G. M. Berns, The FORMAT program, *IEEE Transactions on Engineering Writing & Speech* 11, 2 (August 1968), 85-91.
- [Bigelow 1985] C. Bigelow, Font Design for Personal Workstations, *Byte* 10, 1 (January 1985), 255-270.
- [Biggerstaff et al 1984] T. J. Biggerstaff, E. M. Endres and I. R. Forman, TABLE : Object Oriented Editing of Complex Structures, *Proceedings of the 7th International Conference on Software Engineering*, Orlando, Florida, March 1984, 334-345.
- [Borning 1979] A. Borning, ThingLab - a Constraint-oriented Simulation Laboratory, Report SSL-79-3, Xerox Palo Alto Research Center, 1979.
- [Borning & Duisberg 1986] A. Borning and R. Duisberg, Constraint-based Tools for building User Interfaces, *ACM Transactions on Graphics* 5, 4 (October 1986), 345-374.
- [Brailsford 1986] D. F. Brailsford, Interaction versus Abstraction in the preparation of High-quality Text and Graphics, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 94-97.
- [Brown 1986] P. J. Brown, A Simple Mechanism for Authorship of Dynamic Documents, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 34-42.
- [Bruggemann et al 1986] Bruggemann-Klein, P. Dolland and A. Heinz, How to

- please Authors and Publishers, in *Proceedings of the Second Conference on TeX for Scientific Documentation*, J. Desarmenien (ed.), Springer Verlag, Strasbourg, France, June 1986, 9-31.
- [Bundy 1983] A. Bundy, *The Computer Modelling of Mathematical Reasoning*, Academic Press, 1983.
- [Bundy et al 1979] A. Bundy et al, Mecho: a Program to solve Mechanics problems, Working Paper 50, Department of Artificial Intelligence, University of Edinburgh, 1979.
- [Bush 1945] V. Bush, As we may think, *Atlantic Monthly* 176, 1 (July 1945), 101-108.
- [Carmody et al 1969] S. Carmody, W. Gross, T. H. Nelson, D. Rice and A. van Dam, A Hypertext Editing System for the S/360, in *Pertinent Concepts in Computer Graphics*, J. Nievergelt et al (ed.), University of Illinois Press, Urbana, Illinois, 1969, 291-330.
- [Chamberlin 1987] D. D. Chamberlin, Document Convergence in an Interactive Formatting System, *IBM Journal on Research and Development* 31, 1 (January 1987), 58-72.
- [Chamberlin et al 1982] D. D. Chamberlin et al, JANUS : an Interactive Document Formatter based on Declarative Tags, *IBM System Journal* 21, 3 (1982), 250-271.
- [Chen et al 1986b] P. Chen, J. Coker, M. A. Harrison, J. W. McCarrel and S. Procter, The VorTeX Document Preparation Environment, in *Proceedings of the Second Conference on TeX for Scientific Documentation*, J. Desarmenien (ed.), Springer Verlag, Strasbourg, France, June 1986, 45-54.
- [Chen et al 1986a] P. Chen, J. Coker, M. A. Harrison, J. W. McCarrel and S. Procter, An Improved User Environment for TeX, in *Proceedings of the Second Conference on TeX for Scientific Documentation*, J. Desarmenien (ed.), Springer Verlag, Strasbourg, France, June 1986, 32-44.
- [Chen & Harrison 1988] P. Chen and M. A. Harrison, Multiple Representation Document Development, *IEEE Computer* 21, 1 (January 1988), 15-31.

- [Christodoulakis et al 1986] S. Christodoulakis, F. Ho and M. Theodoridou, The Multimedia Object Presentation Manager of MINOS, in *Proceedings of the International Conference on Management of Data*, Washington, D.C., May 1986, 295-310.
- [Conklin 1987] J. Conklin, Hypertext: an Introduction and Survey, *IEEE Computer* 20, 9 (September 1987), 17-41.
- [Coray et al 1986] G. Coray, R. Ingold and C. Vanoirbeek, Formatting Structured Documents : Batch versus Interactive, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 154-170.
- [Corda & Facchetti 1986] U. Corda and G. Facchetti, Concept Browser : a system for Interactive Creation of Dynamic Documents, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 233-245.
- [Coulouris et al 1976] G. F. Coulouris et al, The Design and Implementation of an Interactive Document Editor, *Software — Practice & Experience* 6, (1976), 271-279.
- [Cowan & Smit 1986] D. D. Cowan and G. V. Smit, Combining Interactive Document Editing with Batch Document Formatting, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 140-153.
- [Cowlshaw 1987] M. F. Cowlshaw, LEXX - a programmable Structured Editor, *IBM Journal on Research and Development* 31, 1 (January 1987), 73-80.
- [Crisanti et al 1985] E. Crisanti, A. Formigoni, G. Gazzano and P. L. Bruna, EasyTeX : an Integrated Environment for Scientific Document Preparation and an Interactive TeX Front End, in *Proceedings of the Second International Conference on Text Processing Systems (PROTEXT II)*, J. J. H. Miller

- (ed.), Dublin, Ireland, October 1985, 145-154.
- [Crisanti et al 1986] E. Crisanti, A. Formigoni and P. L. Bruna, EasyTeX : towards Interactive Formulae Input for Scientific Documents, in *Proceedings of the Second Conference on TeX for Scientific Documentation*, J. Desarmenien (ed.), Springer Verlag, Strasbourg, France, June 1986, 55-64.
- [Czejdo & Embley 1984] B. Czejdo and D. W. Embley, Office Form Definition and Processing using a Relational Data Model, *ACM Notices on Office Automation* 5, 1-2 (June 1984), 123-131.
- [DeSousa 1981] M. R. DeSousa, Electronic Information Interchange in an Office Environment, *IBM System Journal* 20, 1 (1981), 4-23.
- [Delisle & Schwartz 1986] N. Delisle and M. Schwartz, Neptune : a Hypertext system for CAD applications, *Proceedings of the International Conference on Management of Data*, Washington, D.C., May 1986, 132-139.
- [Donzeau-Gouge et al 1984] V. Donzeau-Gouge, B. Lang and B. Melese, A Syntax-directed Program Manipulation Environment, *Proceedings of the 7th International Conference on Software Engineering*, Orlando, Florida, March 1984, 346-354.
- [ECMA 1985] ECMA, Office Document Architecture, Standard ECMA-101, European Computer Manufacturers Association, Geneva, September 1985.
- [Engelbart et al 1973] D. C. Engelbart, R. W. Watson and J. C. Norton, The Augmented Knowledge Workshop, *AFIPS National Computer Conference* 42, (1973), 9-21.
- [Engelbart 1984] D. C. Engelbart, Authorship Provisions in Augment, in *Proceedings of the IEEE Computer Society International Conference (CompCon)*, 1984, 465-472.
- [Fateman 1981] R. J. Fateman, Symbolic and Algebraic Computer Programming Systems, *ACM Bulletin on Symbolic and Algebraic Manipulation* 15, 1 (February 1981), 21-32.
- [Feiner et al 1982] S. Feiner, S. Nagy and A. van Dam, An Experimental System for Creating and Presenting Interactive Graphics,

- ACM Transactions on Graphics* 1, 1 (1982), 59-77.
- [Feiner 1988] S. Feiner, Hierarchical Display for Hypertext Structure, *ACM Bulletin on Office Information System* 9, 2&3 (April 1988), 205-212.
- [Forney 1973] G. D. Forney, The Viterbi Algorithm, *IEEE Proceeding* 61, 3 (March 1973), 268-278.
- [Fraser 1981] C. W. Fraser, Syntax-directed Editing of General Data Structures, in *Proceedings of the ACM Symposium on Text Manipulation*, Portland, Oregon, June 1981, 17-21.
- [Friedell 1984] M. Friedell, Automatic Synthesis of Graphical Object Descriptions, *ACM Computer Graphics* 18, 3 (July 1984), 53-62.
- [Fuchs & Knuth 1985] D. R. Fuchs and D. E. Knuth, Optimal Pre-paging and Font Caching, *ACM Transactions on Programming Languages and Systems* 7, 1 (January 1985), 62-79.
- [Furuta 1986] R. Furuta, An Integrated, but not Exact-representation, Editor/Formatter, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 246-259.
- [Furuta et al 1982] R. Furuta et al, Document Formatting Systems : Survey, Concepts and Issues, *ACM Computing Surveys* 14, 3 (September 1982), 417-472.
- [Futrelle & Barta 1978] R. P. Futrelle and G. Barta, Towards the design of an Intrinsically Graphical Language, *ACM Computer Graphics* 12, 3 (1978), 28-32.
- [Gibbs & Tschritzis 1983] S. Gibbs and D. Tschritzis, A Data Modeling Approach for Office Information Systems, *ACM Transactions on Office Information Systems* 1, 4 (October 1983), 299-319.
- [Goldfarb 1981] C. F. Goldfarb, A Generalized Approach to Document Markup, in *Proceedings of the ACM Symposium on Text Manipulation*, Portland, Oregon, June 1981, 68-73.
- [Goodman 1987] D. Goodman, *The Complete HyperCard Handbook*, Bantam Books, New York, 1987.

- [Gutknecht 1985] J. Gutknecht, Concepts of the Text Editor LARA, *Communications of the ACM* 28, 9 (September 1985), 942-960.
- [Gutknecht & Winiger 1984] J. Gutknecht and W. Winiger, ANDRA : the Document Preparation System of the Personal Workstation Lilith, *Software — Practice & Experience* 14, 1984.
- [Halasz et al 1987] F. G. Halasz, T. P. Moran and R. H. Trigg, NoteCards in a Nutshell, *Proceedings of the International Conference on Human Factors in Computing Systems and Graphics Interface*, Toronto, Canada, April 1987, 45-52.
- [Hamlet 1986] R. Hamlet, A Disciplined Text Environment, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 78-89.
- [Hammer et al 1981] M. Hammer et al, The Implementation of Etude, in *Proceedings of the ACM Symposium on Text Manipulation*, Portland, Oregon, June 1981, 137-146.
- [Horak & Kronert 1984] W. Horak and G. Kronert, An Object-oriented Office Document Architecture Model, *Proceedings of the Second ACM Conference on Office Information Systems*, Toronto, Canada, 1984, 152-160.
- [Horak 1985] W. Horak, Office Document Architecture and Office Document Interchange Formats, *IEEE Computer* 18, 10 (October 1985), 50-60.
- [Hudson & King 1986] S. E. Hudson and R. King, A Generator of Direct Manipulation Office Systems, *ACM Transactions on Office Information Systems* 4, 2 (April 1986), 132-163.
- [ISO 1984] ISO, Vocabulary : Information Processing - Text & Office Systems, Standard 8879, International Organization for Standardization, Geneva, Switzerland, September 1984.
- [ISO 1986] ISO, SGML : Information Processing - Text & Office Systems, Standard 8879, International Organization for Standardization, Geneva, Switzerland, October 1986.
- [Ilson 1980] R. Ilson, An Integrated Approach to Formatted Document

- Production, MIT/LCS/TR-253, Department of Computer Science, MIT, 1980.
- [Johansson et al 1978] S. Johansson, G. N. Leech and H. Goodluck, Manual of Information, Lancaster-Oslo-Bergen (LOB) Corpus of British English, University of Sweden, 1978.
- [Johnson & Beach 1988] J. Johnson and R. J. Beach, Styles in Document Editing Systems, *IEEE Computer* 21, 1 (January 1988), 32-43.
- [Joloboff 1986] V. Joloboff, Trends and Standards in Document Representation, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 107-124.
- [Jones 1986] J. Jones, MACCADD, an enabling Software Method Support Tool, in *Proceedings of the BCS Conference on Human Computer Interaction*, September 1986, 133-154.
- [Katz 1987] A. Katz, Issues in defining an Equation Representation Standard, *ACM Bulletin on Symbolic and Algebraic Manipulation* 21, 2 (May 1987), 19-24.
- [Kay 1986] M. H. Kay, Textmaster - Document filing and retrieving using ODA, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 125-39.
- [Kernighan 1984] B. W. Kernighan, The UNIX Document Preparation Tools - a retrospective, in *Proceedings of the First International Conference on Text Processing Systems (PROTEXT I)*, J. J. H. Miller (ed.), Dublin, Ireland, October 1984, 12-25.
- [Kernighan & Cherry 1975] B. W. Kernighan and L. L. Cherry, A System for Typesetting Mathematics, *Communications of the ACM* 18, 3 (March 1975), 151-156.
- [Kernighan 1981] B. W. Kernighan, Review on 'TeX and MetaFont' - 38151, *ACM Computer Review* 22, 7 (1981), 299-301.
- [Kernighan & Lesk 1982] B. W. Kernighan and M. E. Lesk, UNIX Document Preparation, in *Document Preparation Systems*, J.

- Nievergelt et al (ed.), North Holland, Amsterdam, 1982, 1-19.
- [Kernighan 1982] B. W. Kernighan, PIC - a Language for Typesetting Graphics, *Software — Practice & Experience* 12, (1982), 1-21.
- [Kimura 1984] G. D. Kimura, A Structure Editor and Model for Abstract Document Objects, Technical Report 84-07-04, Computer Science Department, University of Washington, July 1984.
- [Kimura & Shaw 1984] G. D. Kimura and A. C. Shaw, The Structure of Abstract Document Objects, *ACM Notices on Office Automation* 5, 1-2 (June 1984), 161-169.
- [Kitagawa et al 1984] H. Kitagawa, M. Gotoh, S. Misaki and M. Azuma, Form Document Management System SPECDOQ, *ACM Notices on Office Automation* 5, 1-2 (June 1984), 132-142.
- [Knuth 1978] D. E. Knuth, Mathematical Typography, STAN-CS-78-648, Computer Science Department, Stanford University, February 1978.
- [Knuth 1979] D. E. Knuth, in *TeX and METAFONT : New Directions in Typesetting*, American Mathematical Society & Digital Press, 1979.
- [Knuth & Plass 1980] D. E. Knuth and M. F. Plass, Breaking Paragraphs into Lines, STAN-CS-80-828, Computer Science Department, Stanford University, November 1980.
- [Knuth 1983] D. E. Knuth, The WEB System for Structured Documentation, STAN-CS-83-980, Computer Science Department, Stanford University, September 1983.
- [Knuth 1984] D. E. Knuth, *The TeXbook*, Addison Wesley, Reading, Massachusetts, 1984.
- [Lamport 1986] L. Lamport, *LaTeX : a document preparation system*, Addison Wesley, Reading, Massachusetts, 1986.
- [Lampson 1978] B. W. Lampson, *Bravo Manual (in Alto User's Handbook)*, Xerox Palo Alto Research Center, 1978.
- [LeVan 1985] H. LeVan, SGML : a Standard Language for Text

- Description, in *Proceedings of the Second International Conference on Text Processing Systems (PROTEXT II)*, J. J. H. Miller (ed.), Dublin, Ireland, October 1985, 198-212.
- [LeVan & Terreni 1986] H. LeVan and E. Terreni, A Language to describe Formatting Directives for SGML Documents, in *Proceedings of the Second Conference on TeX for Scientific Documentation*, J. Desarmenien (ed.), Springer Verlag, Strasbourg, France, June 1986, 98-119.
- [Leler & Soiffer 1985] W. Leler and N. Soiffer, An Interactive Graphical Interface for REDUCE, *ACM Bulletin on Symbolic and Algebraic Manipulation* 19, 3 (August 1985), 17-23.
- [Lesk 1979] M. E. Lesk, TBL - a Program to Format Tables, UNIX Programmer's Manual, AT & T Bell Laboratories, 1979.
- [Levison 1983] M. Levison, Editing Mathematical Formulae, *Software — Practice & Experience* 13, 2 (February 1983), 189-195.
- [Lipkie et al 1982] D. E. Lipkie, S. R. Evans, J. K. Newlin and L. Weissman, Star Graphics : an Object-oriented Implementation, *ACM Computer Graphics* 16, 3 (July 1982), 115-124.
- [Mackinlay 1986] J. Mackinlay, Automating the Design of Graphical Presentations of Relational Information, *ACM Transactions on Graphics* 5, 2 (April 86), 110-141.
- [Madnick et al 1968] S. E. Madnick and A. Moulton, Script, an On-line Manuscript Processing System, *IEEE Transactions on Engineering Writing & Speech* 11, 2 (August 1968), 92-100.
- [Martin 1967] W. A. Martin, Symbolic Mathematical Laboratory, MAC-TR-36, Computer Science Department, MIT, January 1967.
- [MathSoft 1987] MathSoft, *MathCAD Use Manual*, MathSoft Ltd., Massachusette, 1987.
- [Maxemchuch & Wilder 1981] N. F. Maxemchuch and H. A. Wilder, Virtual Editing: the concept, in *Document Preparation Systems*, J. Nievergelt et al (ed.), North Holland, Amsterdam, 1981.
- [McCracken & Akscyn 1984] D. McCracken and R. M. Akscyn, Experience with the ZOG Human Computer Interface, *International Journal on*

- Man Machine Studies* 21, 2 (August 1984), 293-310.
- [Medina-Mora 1982] R. Medina-Mora, Syntax Directed Editing : towards Integrated Programming Environment, CMU-CS-82-113 , Department of Computer Science, Carnegie-Mellon University, March 1982.
- [Meyrowitz & van Dam 1982] N. Meyrowitz and A. van Dam, Interactive Editing Systems, *ACM Computing Surveys* 14, 3 (September 1982), 321-415.
- [Microsoft 1986] Microsoft, *Microsoft Word 3.0 User Manual*, Microsoft Corp., 1986.
- [Morris 1985] R. A. Morris, Interleaf : Is What you See Enough to Get ?, in *Proceedings of the Second International Conference on Text Processing Systems (PROTEXT II)*, J. J. H. Miller (ed.), Dublin, Ireland, October 1985, 56-81.
- [Naffah & Karmouch 1986] N. Naffah and A. Karmouch, AGORA - an experiment in Multimedia Message System, *IEEE Computer* 19, 5 (May 1986), 56-66.
- [Nelson 1985] G. Nelson, Juno, a Constraint-based Graphics System, *ACM Computer Graphics* 19, 3 (July 1985), 235-243.
- [Neuhoff 1975] D. L. Neuhoff, The Viterbi Algorithm as an aid in Text Recognition, *IEEE Transactions on Information Theory* 21, 2 (1975), 222-226.
- [Olsen 1983] D. R. Olsen, Automatic Generation of Interactive System, *ACM Computer Graphics* 17, 1 (January 1983), 53-57.
- [Ossanna 1977] J. F. Ossanna, TROFF User's Manual, Computing Science Technical Report No.54, AT & T Bell Laboratories, 1977.
- [Peels et al 1985] A. J. H. M. Peels, N. J. M. Janssen and W. Nawijn, Document Architecture and Text Formatting, *ACM Transactions on Office Information Systems* 3, 4 (October 1985), 347-369.
- [Plass 1981] M. F. Plass, Optimal Pagination Techniques for Automatic Typesetting Systems, STAN-CS-81-870, Computer Science Department, Stanford University, June 1981.

- [Poggio et al 1985] A. Poggio et al, CCWS: a Computer-based, Multimedia Information System, *IEEE Computer* 18, 10 (October 1985), 92-103.
- [Purvy et al 1983] R. Purvy, J. Farrell and P. Klose, The Design of Star's Record Processing, *ACM Transactions on Office Information Systems* 1, 1 (January 1983), 3-24.
- [Quint 1983] V. Quint, An Interactive System for Mathematical Text Processing, *Technology & Science of Informatics* 2, 3 (1983), 169-179.
- [Quint 1984] V. Quint, Interactive Editing of Mathematics, in *Proceedings of the First International Conference on Text Processing Systems (PROTEXT 1)*, J. J. H. Miller (ed.), Dublin, Ireland, October 1984, 55-68.
- [Quint 1985] V. Quint, Towards Interactive Math Mode in TeX, in *Proceedings of the First Conference on TeX for Scientific Documentation*, D. Lucarella (ed.), Addison Wesley, Reading, Massachusetts, June 1985, 21-35.
- [Quint et al 1986a] V. Quint and H. Bedor, GRIF : an Interactive Environment for TeX, in *Proceedings of the Second Conference on TeX for Scientific Documentation*, J. Desarmenien (ed.), Springer Verlag, Strasbourg, France, June 1986, 145-158.
- [Quint et al 1986b] V. Quint and I. Vatton, GRIF : an Interactive System for Structured Document Manipulation, in *Proceedings of the International Conference on Text Processing and Document Manipulation*, J. C. van Vliet (ed.), Cambridge University Press, Nottingham, England, April 1986, 200-213.
- [Reid 1980a] B. K. Reid, SCRIBE : a Document Specification Language and its Compiler, CMU-CS-81-100, Computer Science Department, Carnegie-Mellon University, October 1980.
- [Reid 1980b] B. K. Reid, A High-level Approach to Computer Document Formatting, in *Proceedings of the 7th ACM Symposium on Principles of Programming Language*, January 1980, 24-31.
- [Rowe et al 1987] L. Rowe et al, A Browser for Directed Graphs, *Software —*

- Practice & Experience* 17, 1 (January 87), 61-76.
- [Saltzer 1965] J. Saltzer, Manuscript Typing and Editing : TYPSET, RUNOFF, in *The Compatible Time-sharing System*, P. A. Crisman (ed.), MIT Press, 1965.
- [Schich 1982] T. Schick, The Document Interchange Architecture (DIA), *IBM System Journal* 21, 2 (1982), 220-246.
- [Seybold 1985] J. W. Seybold, Generic Coding and WYSIWYG Displays, *Seybold Report on Publishing Systems* 14, 11 (February 1985), p.19.
- [Seybold 1987] J. W. Seybold, The Desktop Publishing Phenomenon, *Byte* 12, 5 (May 1987), 149-154.
- [Shannon 1951] C. E. Shannon, Prediction and Entropy of Printed English, *Bell System Technical Journal* 30, (1951), 50-64.
- [Shaw 1980] A. C. Shaw, A Model for Document Preparation System, Technical Report 80-04-02, Department of Computer Science, University of Washington, April 1980.
- [Smith et al 1982a] D. C. Smith, C. Irby, R. Kimball, B. Verplank and E. Harslem, Designing the Star User Interface, *Byte* 7, 4 (April 1982), 242-282.
- [Smith et al 1982b] D. C. Smith, C. Irby and R. Kimball, The Star User Interface : an Overview, *AFIPS National Computer Conference* 51, (1982), 515-528.
- [Stallman 1981] R. M. Stallman, EMACS : the Extensible, Customizable Self-documenting Display Editor, in *Proceedings of the ACM Symposium on Text Manipulation*, Portland, Oregon, June 1981, 147-156.
- [Stroemfors & Jonesjoe 1981] O. Stroemfors and L. Jonesjoe, The Implementation and Experience of a Structure-oriented Text Editor, in *Proceedings of the ACM Symposium on Text Manipulation*, Portland, Oregon, June 1981, 22-27.
- [Suen 1979] C. Y. Suen, N-gram Statistics for Natural Language Understanding and Text Processing, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1, 2 (April 1979), 164-172.

- [Sutherland 1963] I. E. Sutherland, SketchPad - a Man/machine Graphical Communication System, *AFIPS Spring Joint Computer Conference* 23, (1963), 329-346.
- [Teitelbaum & Reps 1981] T. Teitelbaum and T. Reps, The Cornell Program Synthesizer, *Communications of the ACM* 24, 9 (September 1981), 563-573.
- [Teitelman 1983] W. Teitelman, The Cedar Programming Environment, Report CSL-83-11, Xerox Palo Alto Research Center, December 1983.
- [Thomas 1983] J. J. Thomas, Graphical Input Interaction Technique, *ACM Computer Graphics* 17, 1 (January 1983), 5-30.
- [Trevorrow 1986] A. Trevorrow, DVI-TO-VDU : a TeX Page Previewer, *TUGBOAT* 7, 1 (March 1986), 25-36.
- [Trigg & Weiser 1986] R. H. Trigg and M. Weiser, TEXTNET : a Network-based approach to Text Handling, *ACM Transactions on Office Information Systems* 4, 1 (January 1986), 1-23.
- [Unilogic 1983] Unilogic, *The Scribe User Manual*, Unilogic Ltd., Pittsburgh, Pennsylvania, 1983.
- [van Wyk 1981] C. J. van Wyk, IDEAL : a Graphics Typesetting Language, *ACM Notices on Programming Language* 16, 6 (June 1981), 99-107.
- [Walker 1981] J. H. Walker, The Document Editor, in *Proceedings of the ACM Symposium on Text Manipulation*, Portland, Oregon, June 1981, 44-50.
- [Walker 1988] J. H. Walker, Supporting Document Development with Concordia, *IEEE Computer* 21, 1 (January 1988), 48-59.
- [Warnock 1980] J. E. Warnock, The Display of Characters using Gray Level Sample Arrays, Report CSL-80-6, Xerox Palo Alto Research Center, May 1980.
- [Warnock & Wyatt 1982] J. E. Warnock and D. K. Wyatt, A Device Independent Graphics Imaging Model for Raster Devices, *ACM Computer Graphics* 16, 3 (1982), .
- [Weyer & Borning 1985] S. A. Weyer and A. H. Borning, A prototype Electronic

- Encyclopedia, *ACM Transactions on Office Information Systems* 3, 1 (January 1985), 63-68.
- [Wilkes & Wiseman 1982] A. J. Wilkes and N. E. Wiseman, A Soft-edged Character Set and its Derivation, *The Computer Journal* 25, 1 (January 1982), 140-147.
- [Wiseman et al 1972] N. E. Wiseman, C. J. Cheney, M. Etherton, J. O. Hiles and H. U. Lemke, Rainbow - a Multi-purpose CAD System, *Software — Practice & Experience* 2, (1972), 359-375.
- [Wiseman 1973] N. E. Wiseman, A Computer Graphics System written in BCPL, *INRIA-AFCET Colloques. Journees Graphique* 1, (1973), 77-85.
- [Wiseman et al 1978] N. E. Wiseman et al, On making Graphic Arts Quality Output by Computer, *The Computer Journal* 21, 1 (February 1978), 2-6.
- [Wiseman et al 1979] N. E. Wiseman et al, Aspects of Quality in the Design and Production of Text, *ACM Computer Graphics* 13, 2 (February 1979), 63-67.
- [Witten 1984] I. H. Witten, Dynamic Documents, in *Proceedings of the First International Conference on Text Processing Systems (PROTEXT I)*, J. J. H. Miller (ed.), Dublin, Ireland, October 1984, 234-239.
- [Yankelovich & Meyrowitz 1985] N. Yankelovich and N. Meyrowitz, Reading and Writing the Electronic Book, *IEEE Computer* 18, 10 (October 1985), 15-30.
- [Yankelovich et al 1988] N. Yankelovich, B. J. Hann, N. K. Meyrowitz and S. M. Drucker, Intermedia - a Seamless Information Environment, *IEEE Computer* 21, 1 81-96.
- [Yao et al 1984] S. B. Yao, A. R. Hevner, Z. Shi and D. Luo, Formanager : an Office Forms Management System, *ACM Transactions on Office Information Systems* 2, 3 (July 1984), 235-262.

CONTENTS

Final References 211

10.1 Document Modeling 211

10.2 Hypertext Systems 211

10.3 Structure Editing 212

10.4 Batch Systems 212

10.5 Interactive Systems 212

10.6 User Interface Generation 212

10.7 Interactive or Wysiwyg 213

10.8 Equations 213

10.9 Tables and Forms 213

10.10 Graphics Editing 213

10.11 Font Caching 214

10.12 Misellaneous 214

APPENDIX A

Overall Document Architecture

Composite objects (§3.2.1.2) :

```
<document>          ::=  <book>
                        | <report>
                        | <letter>
                        | <thesis>
                        | <manual>
```

This part is specific to the document type 'book' :

```
<book>               ::=  <front matter> , <main body> , <back matter>

<front matter>       ::=  <title page> + <preface> + <introduction> + <contents>

<title page>         ::=  <book title> + <date> + <author list>
<book title>         ::=  <text>
<date>               ::=  <text>
<author list>        ::=  { <author name> + [<affiliation>] }
<author name>        ::=  <text>
<affiliation>        ::=  <text>

<preface>            ::=  { <text> }
<introduction>       ::=  { <text> }

<contents>           ::=  { <title> + <page number> }
<title>              ::=  <text>
<page number>        ::=  <text>
```

```
<main body>          ::=  { <chapter> }
```

<chapter>	::= <chapter title> + (<content> , {<section>})
<chapter title>	::= <text>
<section>	::= <section title> + (<content> , {<section>})
<section title>	::= <text>
<content>	::= <main content> + <auxiliary content>
<main content>	::= { <elementary object> <local composite object> }
<elementary object>	::= (see below)
<local composite object>	::= <quotation> <enumerated list> <worked example> <theorem>
<quotation>	::= <quoted text> + <quote source>
<quoted text>	::= { <elementary object> <local composite object> }
<quote source>	::= <text>
<enumerated list>	::= { <list element> }
<list element>	::= <elementary object> <local composite object>
<worked example>	::= <problem> + <solution>
<problem>	::= { <elementary object> <local composite object> }
<solution>	::= { <elementary object> <local composite object> }
<auxiliary content>	::= <footnote object> <floating object>
<footnote object>	::= { <text> }
<floating object>	::= <caption> + <exhibit>
<caption>	::= <text>
<exhibit>	::= { <elementary object> }
<back matter>	::= <bibliography> + <index> + {<appendix>}
<bibliography>	::= { <reference object> }
<reference object>	::= {<author name>} + <article title> + <reference source>
<author name>	::= <text>
<article title>	::= <text>
<reference source>	::= <book source> <journal source>
<book source>	::= <book title> + <publisher> + <date>
<journal source>	::= <journal title> + <institution> + <date>

		+ <journal volume> + <journal number>
<book title>	::=	<text>
<publisher>	::=	<text>
<date>	::=	<text>
<journal title>	::=	<text>
<institution>	::=	<text>
<journal volume>	::=	<text>
<journal number>	::=	<text>
<index>	::=	{ <index object> }
<index object>	::=	<indexed keyword> + {<page number>}
<indexed keyword>	::=	<text>
<page number>	::=	<text>
<appendix>	::=	<appendix title> + (<content> , {<subappendix>})
<appendix title>	::=	<text>
<subappendix>	::=	<subappendix title> + (<content> , {<subappendix>})
<subappendix title>	::=	<text>

'book' specific definitions terminate here

Elementary objects (§3.2.1.1):

<elementary object>	::=	<unstructured object> <structured object>
<unstructured object>	::=	<image> <unstructured diagram> (see §3.2.1.1.2)
<structured object>	::=	<text> <program> <equation> <table> <structured diagram>
<text>	::=	{ <sentence> }
<sentence>	::=	{ <equation> <word> }
<word>	::=	{ <character> }
<character>	::=	'a' 'b' 'c' ...
<program>	::=	(syntax definition of language in question)
<equation>	::=	(see Appendix B.1)
<table>	::=	(see Appendix D.1)
<structured diagram>	::=	<tree-based diagram>

| <network-based diagram>
 | <data-driven diagram>
 (see discussions in §6.2)

Non-hierarchical links (§3.2.1.3) :

<cross reference link> ::= <text> → (<composite object>|<elementary object>)

<content sharing link> ::= <elementary object> → <elementary object>
 | <composite object> → <composite object>

<floating object anchor> ::= <text> → <floating object>

<footnote link> ::= <text> → <footnote object>

<index link> ::= <text> → <index object>

<bibliography link> ::= <text> → <reference object>

Writing support (§3.2.2.1) :

<idea pool> ::= {<composite object> | <elementary object>}

<consistency link> ::= (<composite object> | <elementary object>)
 → (<composite object> | <elementary object>)

Reading support (§3.2.2.2) :

<notepad> ::= {<composite object> | <elementary object>}

<relevancy link> ::= (<composite object> | <elementary object>)
 → (<composite object> | <elementary object>)

<reading path> ::= (<composite object> | <elementary object>)
 → (<composite object> | <elementary object>)

APPENDIX B

Architecture for Equations

B.1 Logical model

<equation>	::=	<relation> <expression>
<relation>	::=	<quantified relation> <unquantified relation>
<quantified relation>	::=	{ <quantifier> } + <unquantified relation>
<quantifier>	::=	<universal quantifier> <existential quantifier>
<universal quantifier>	::=	<literal expression>
<existential quantifier>	::=	<literal expression>
<unquantified relation>	::=	<comparative relation> <general relation> <set relation> <logic relation>
<comparative relation>	::=	<equality relation> <inequality relation> <greater-than relation> <greater/equal relation> <less-than relation> <less/equal relation>
<equality relation>	::=	<left expression> + <right expression>
<inequality relation>	::=	<left expression> + <right expression>
<greater-than relation>	::=	<left expression> + <right expression>
<greater/equal relation>	::=	<left expression> + <right expression>
<less-than relation>	::=	<left expression> + <right expression>

<less/equal relation> ::= <left expression> + <right expression>

<left expression> ::= <expression>

<right expression> ::= <expression>

<general relation> ::= <approach relation>
 | <proportional relation>
 | <definition relation>
 | <similarity relation>
 | <approximate relation>
 | <identity relation>

<approach relation> ::= <left expression> + <right expression>

<proportional relation> ::= <left expression> + <right expression>

<definition relation> ::= <left expression> + <right expression>

<similarity relation> ::= <left expression> + <right expression>

<approximate relation> ::= <left expression> + <right expression>

<identity relation> ::= <left expression> + <right expression>

<left expression> ::= <expression>

<right expression> ::= <expression>

<set relation> ::= <subset relation>
 | <superset relation>
 | <improper-subset relation>
 | <improper-superset relation>
 | <reflex-subset relation>
 | <reflex-superset relation>
 | <member relation>

<subset relation> ::= <left expression> + <right expression>

<superset relation> ::= <left expression> + <right expression>

<improper-subset relation> ::= <left expression> + <right expression>

<improper-superset relation> ::= <left expression> + <right expression>

<reflex-subset relation> ::= <left expression> + <right expression>

<reflex-superset relation> ::= <left expression> + <right expression>

<member relation> ::= <left expression> + <right expression>

<left expression> ::= <expression>

<right expression> ::= <expression>

<logic relation> ::= <implication relation>
 | <double implication relation>
 | <equivalence relation>
 | <congruence relation>
 | <deduction relation>
 | <consequence relation>

<implication relation>	::=	<left expression> + <right expression>
<double implication relation>	::=	<left expression> + <right expression>
<equivalence relation>	::=	<left expression> + <right expression>
<congruence relation>	::=	<left expression> + <right expression>
<deduction relation>	::=	<left expression> + <right expression>
<consequence relation>	::=	<left expression> + <right expression>

<left expression>	::=	<expression> <relation>
<right expression>	::=	<expression> <relation>

<expression>	::=	<terminal expression> <unary expression> <binary expression> <n-ary expression>
--------------	-----	--

<terminal expression>	::=	<literal expression> <numeric expression>
-----------------------	-----	--

<literal expression>	::=	<base identifier> + [<script>]
<base identifier>	::=	<arbitrary string>
<script>	::=	<arbitrary string> (not further decomposed)

<numeric expression>	::=	<numeric string> (not further decomposed)
----------------------	-----	--

<unary expression>	::=	<trigonometric function> <hyperbolic function> <unary scalar function> <unary vector function>
--------------------	-----	---

<trigonometric function>	::=	<sine function> <cosine function> <tangent function> <inverse-sin function> <inverse-cos function> <inverse-tan function>
--------------------------	-----	--

<sine function>	::=	<expression>
<cosine function>	::=	<expression>
<tangent function>	::=	<expression>
<inverse-sin function>	::=	<expression>
<inverse-cos function>	::=	<expression>
<inverse-tan function>	::=	<expression>

<hyperbolic function>	::=	<sinh function> <cosh function> <tanh function> <inverse sinh function> <inverse cosh function> <inverse tanh function>
-----------------------	-----	--

<sinh function>	::=	<expression>
<cosh function>	::=	<expression>
<tanh function>	::=	<expression>
<inverse sinh function>	::=	<expression>
<inverse cosh function>	::=	<expression>
<inverse tanh function>	::=	<expression>

<unary scalar function>	::=	<factorial function> <ceiling function> <floor function> <absolute function> <modulus function> <argument function> <negation function> <exponential function> <logarithmic function> <root function> <reciprocal function>
-------------------------	-----	---

<factorial function>	::=	<expression>
<ceiling function>	::=	<expression>
<floor function>	::=	<expression>
<absolute function>	::=	<expression>
<modulus function>	::=	<expression>
<argument function>	::=	<expression>
<negation function>	::=	<expression>
<exponential function>	::=	<expression>
<logarithmic function>	::=	<expression>
<root function>	::=	<expression>
<reciprocal function>	::=	<expression>

<unary vector function>	::=	<transpose function> <inverse function> <gradient function> <laplacian function> <euclidistance function> <norm function> <curl function>
-------------------------	-----	---

<transpose function>	::=	<expression>
<inverse function>	::=	<expression>

<gradient function>	::=	<expression>
<laplacian function>	::=	<expression>
<euclidistance function>	::=	<expression>
<norm function>	::=	<expression>
<curl function>	::=	<expression>

<binary expression>	::=	<binary scalar operation> <binary logic operation> <binary set operation>
---------------------	-----	---

<binary scalar operation>	::=	<addition operation> <subtraction operation> <add/minus operation> <multiplication operation> <division operation> <power operation>
---------------------------	-----	---

<addition operation>	::=	<left operand> + <right operand>
<subtraction operation>	::=	<left operand> - <right operand>
<add/minus operation>	::=	<left operand> + <right operand>
<multiplication operation>	::=	<left operand> * <right operand>
<division operation>	::=	<left operand> / <right operand>
<power operation>	::=	<left operand> ^ <right operand>
<complex expression>	::=	<left operand> + <right operand>

<left operand>	::=	<expression>
<right operand>	::=	<expression>

<binary logic operation>	::=	<and operation> <or operation> <nand operation> <nor operation> <exclusive-and operation> <exclusive-or operation>
--------------------------	-----	---

<and operation>	::=	<left operand> & <right operand>
<or operation>	::=	<left operand> <right operand>
<nand operation>	::=	<left operand> & <right operand>
<nor operation>	::=	<left operand> <right operand>
<exclusive-and operation>	::=	<left operand> & <right operand>
<exclusive-or operation>	::=	<left operand> <right operand>
<complex expression>	::=	<left operand> + <right operand>

<left operand>	::=	<expression>
<right operand>	::=	<expression>

<binary set operation>	::=	<union operation> <intersect operation>
<union operation>	::=	<left operand> + <right operand>
<intersect operation>	::=	<left operand> + <right operand>
<left operand>	::=	<expression>
<right operand>	::=	<expression>
<n-ary expression>	::=	<matrix expression> <series expression> <calculus expression> <polynomial expression> <limit expression> <user defined function>
<matrix expression>	::=	<row-major matrix> <column-major matrix>
<row-major matrix>	::=	{ <matrix row> }
<matrix row>	::=	{ <matrix element> }
<column-major matrix>	::=	{ <matrix column> }
<matrix column>	::=	{ <matrix element> }
<matrix element>	::=	<expression>
<series expression>	::=	<summation expression> <product expression> <union expression> <intersect expression>
<summation expression>	::=	<summand> + <index> + [<lower bound> + <upper bound>]
<summand>	::=	<expression>
<product expression>	::=	<productand> + <index> + [<lower bound> + <upper bound>]
<productand>	::=	<expression>
<intersection expression>	::=	<main expression> + <index> + [<lower bound> + <upper bound>]
<union expression>	::=	<main expression> + <index> + [<lower bound> + <upper bound>]
<main expression>	::=	<expression>
<index>	::=	<expression>
<lower bound>	::=	<expression>
<upper bound>	::=	<expression>

<calculus expression>	::=	<integral expression> <derivative expression>
<integral expression>	::=	<definite integral> <indefinite integral>
<definite integral>	::=	<integrand> + <reference> + <lower bound> + <upper bound>
<indefinite integral>	::=	<integrand> + <reference>
<integrand>	::=	<expression>
<reference>	::=	<expression>
<lower bound>	::=	<expression>
<upper bound>	::=	<expression>
<derivative expression>	::=	<main expression> + <reference> + [<order>]
<main expression>	::=	<expression>
<reference>	::=	<expression>
<order>	::=	<expression>

<polynomial expression>	::=	{ <component expression> }
<component expression>	::=	<coefficient> + <base expression> + <order>
<base expression>	::=	{expression}
<coefficient>	::=	{expression}
<order>	::=	{expression}

<limit expression>	::=	<main expression> + <control variable> + <limit>
<main body>	::=	<expression>
<control variable>	::=	<expression>
<limit>	::=	<expression>

<user defined function>	::=	<function identifier> + { <parameter> }
<function identifier>	::=	<literal expression>
<parameter>	::=	<expression>

B.2 Physical Model

<math box>	::=	<symbol box> <special box> <linked box> <scripted box> <over box> <array box>
<special box>	::=	<delimited box> <accented box> <rooted box>
<delimited box>	::=	<delimiter type> + <math box>
<delimiter type>	::=	<parenthesis box> <bracket> <brace> <angle bracket> <floor> <ceiling> <vertical bar> <double vertical bar>
<accented box>	::=	<accent type> + <math box>
<accent type>	::=	<dot> <di-dot> <tilde> <bar> <hat> <check> <acute> <grave> <breve> <vector>
<rooted box>	::=	<big root sign> + <math box>

<linked box>	::=	<horizontal list> <vertical list>
<horizontal list>	::=	{ <math box> }
<vertical list>	::=	{ <math box> }

<scripted box>	::=	<superscript box> <subscript box> <super-subscript box> <pre-superscript box> <pre-subscript box> <pre-super-subscript box>
<superscript box>	::=	<base box> + <script box>
<subscript box>	::=	<base box> + <script box>
<super-subscript box>	::=	<base box> + <script box> + <script box>
<pre-superscript box>	::=	<base box> + <script box>
<pre-subscript box>	::=	<base box> + <script box>
<pre-super-subscript box>	::=	<base box> + <script box> + <script box>
<base box>	::=	<math box>
<script box>	::=	<math box>

<over box>	::=	<top box> + <bottom box>
<top box>	::=	<math box>
<bottom box>	::=	<math box>

<array box>	::=	{ <row box> }
<row box>	::=	{ <element box> }
<element box>	::=	<math box>

B.3 Formatting Rules

(Only the rules for implemented templates are listed here.)

Relations :

Equality (left expression, right expression)

→ Horiz (left expression, '=', right expression)

Inequality (left expression, right expression)

→ Horiz (left expression, '≠', right expression)

Less-than (left expression, right expression)

→ Horiz (left expression, '<', right expression)

Less/equal (left expression, right expression)

→ Horiz (left expression, '≤', right expression)

Greater-than (left expression, right expression)

→ Horiz (left expression, '>', right expression)

Greater/equal (left expression, right expression)

→ Horiz (left expression, '≥', right expression)

Approximation (left expression, right expression)

→ Horiz (left expression, '≈', right expression)

Similarity (left expression, right expression)

→ Horiz (left expression, '∼', right expression)

Identity (left expression, right expression)

→ Horiz (left expression, '≡', right expression)

Proportionality (left expression, right expression)

→ Horiz (left expression, ' $\propto$ ', right expression)

Approach (left expression, right expression)

→ Horiz (left expression, ' $\rightarrow$ ', right expression)

Definition (left expression, right expression)

→ Horiz (left expression, Vert ('*def*', '='), right expression)

Subset (left expression, right expression)

→ Horiz (left expression, ' $\subset$ ', right expression)

Improper-subset (left expression, right expression)

→ Horiz (left expression, ' $\subseteq$ ', right expression)

Superset (left expression, right expression)

→ Horiz (left expression, ' $\supset$ ', right expression)

Improper-superset (left expression, right expression)

→ Horiz (left expression, ' $\supseteq$ ', right expression)

Member (left expression, right expression)

→ Horiz (left expression, ' $\in$ ', right expression)

Unary expressions :

Sine (expression)

→ Horiz ('sin', Delimit-Paranthesis (expression))

Cosine (expression)

→ Horiz ('cos', Delimit-Paranthesis (expression))

Tangent (expression)

→ Horiz ('tan', Delimit-Paranthesis (expression))

Inverse-sine (expression)

→ Horiz (Superscript ('sin', '-1'), Delimit-Paranthesis (expression))

Inverse-cosine (expression)

→ Horiz (Superscript ('cos', '-1'), Delimit-Paranthesis (expression))

Inverse-tangent (expression)

→ Horiz (Superscript ('tan', '-1'), Delimit-Paranthesis (expression))

Sinh (expression)

→ Horiz ('sinh', Delimit-Paranthesis (expression))

Cosh (expression)

→ Horiz ('cosh', Delimit-Paranthesis (expression))

Tanh (expression)

→ Horiz ('tanh', Delimit-Paranthesis (expression))

Inverse-sinh (expression)

→ Horiz (Superscript ('sinh', '-1'), Delimit-Paranthesis (expression))

Inverse-cosh (expression)

→ Horiz (Superscript ('cosh', '-1'), Delimit-Paranthesis (expression))

Inverse-tanh (expression)

→ Horiz (Superscript ('tanh', '-1'), Delimit-Paranthesis (expression))

Root (expression)

→ Delimit-Root (expression)

Factorial (expression)

→ Horiz (expression, '!')

Inverse (expression)

→ Superscript (expression, '-1')

Reciprocal (expression)

→ Superscript (expression, '-1')

→ Over ('1', expression)

Negation (expression)

→ Horiz ('-', expression)

Ceiling (expression)

→ Delimiter-Ceiling (expression)

Floor (expression)

→ Delimiter-Floor (expression)

Natural logarithm (expression)

→ Horiz ('ln', expression)

Exponential (expression)

→ Superscript ('e', expression)

Logarithm (expression)

→ Horiz ('log', expression)

Logical-not (expression)

→ Horiz ('¬', expression)

Binary expressions :

Addition (left expression, right expression)

→ Horiz (left expression, '+', right expression)

Subtraction (left expression, right expression)

→ Horiz (left expression, '-', right expression)

Add-minus (left expression, right expression)

→ Horiz (left expression, '±', right expression)

Multiplication (left expression, right expression)

→ Horiz (left expression, '×', right expression)

Division (left expression, right expression)

→ Horiz (left expression, '÷', right expression)

→ Horiz (left expression, '/', right expression)

→ Over (left expression, right expression)

Union (left expression, right expression)

→ Horiz (left expression, 'U', right expression)

Intersection (left expression, right expression)

→ Horiz (left expression, '∩', right expression)

Power (left expression, right expression)

→ Superscript (left expression, right expression)

Subscript (left expression, right expression)

→ Subscript (left expression, right expression)

Complex-orthogonal (left expression, right expression)

→ Horiz (left expression, '+ j', right expression)

Complex-exponential (left expression, right expression)

→ Superscript ('e', Horiz (left expression, '+ j', right expression))

Complex-polar (left expression, right expression)

→ Horiz (left expression, Superscript ('e', Horiz ('j', right expression)))

n-nary expressions :

Limit (main expression, variable, limit)

→ Horiz (Vert ('lim', Horiz (variable, '→', limit), main expression)

Summation (summand, index, lower bound, upper bound)

→ Horiz (Vert (upper bound, 'Σ', Horiz (index, '=', lower bound)), summand)

→ Horiz (Vert ('Σ', Horiz (lower bound, '≤', index, '≤', upper bound)), summand)

→ Horiz (Vert ('Σ', index) , summand)

Product (main expression, index, lower bound, upper bound)

→ Horiz (Vert (upper bound, 'Π', Horiz (index, '=', lower bound)), main expression)

→ Horiz (Vert ('Π', Horiz (lower bound, '≤', index, '≤', upper bound)), main expression)

→ Horiz (Vert ('Π', index) , main expression)

Definite Integral (integrand, variable, lower bound, upper bound)

→ Horiz (Vert (upper bound, '∫', lower bound), integrand, 'd', variable)

Indefinite Integral (integrand, variable)

→ Horiz ('∫', integrand, 'd', variable)

Partial-derivative-2 (expression, reference)

→ Over (Horiz (Superscript('∂','2'),expression),Horiz ('∂', Superscript (reference,'2'))

Matrices :

Matrix (matrix row, ... , matrix row)

→ Array (matrix row, ... , matrix row)

Matrix-row (matrix element, ... , matrix element)

→ Horiz (matrix element, ... , matrix element)

APPENDIX C

The Equation Editor

C.1 List of Commands

Cursor movements :

A user is at a particular node of an equation tree at any time. Depending on whether this *cursor node* is a leaf, either an expression or a single symbol is selected, both being referred to as the *cursor* here. A selected expression is highlighted with a red box around it, while a selected symbol is displayed in red. There are four methods of selection, all using the left mouse button.

1. **Single click** selects an individual symbol, provided the pointer is directed at one. With the shift key depressed, however, a single click selects the entire equation directly, irrespective of the pointer's location.
2. **Double click** helps to move up or down an equation tree. It selects either :
 - the *parent* expression of the cursor, if the pointer lies *outside* the latter, or
 - a *descendent* expression of the cursor, if the pointer lies *inside* the latter; in this case, the pointer's location determines which descendent is selected.
3. **Dragging** with button depressed invokes a rubber box. It is used to select directly any arbitrary expression, which is the largest logical unit being totally enclosed. The entire equation can be selected in this way as well (apart from a shifted single click).

Editing Commands :

Some commands are active only when a symbol is selected, while some works for expressions only. These are marked 'leaf only' and 'non-leaf only' in the command list that

follows. Invalid options are disabled accordingly as the user browses around an equation. Commands are organized into the following categories, each being accessible through one drop-down menu.

File :

1. **New.** The existing equation is erased and replaced by a metabox.
2. **Open.** (Leaf only.) An expression is loaded from a specified file and plugged into the cursor symbol.
3. **Save.** The entire equation is saved to a specified file.
4. **Save cursor.** (Non-leaf only) The cursor expression is saved to a specified file.
5. **Print.** A GEM metafile is created for the entire equation.
6. **Quit.** The editing session is quit.

Edit :

1. **Append box.** A box symbol is appended to the cursor according to one of the following options.
 - The new box being placed *before*, *after*, *above* or *below* the cursor.
 - The new box being a *subscript*, *superscript* or simultaneous *super-subscripts* to the cursor. Two boxes are created in the latter case.
 - The new box being either the *numerator* or *denominator* of a fraction formed with the cursor.

If the cursor is a row in a matrix, only the 'above' and 'below' options are valid; in this case, a row of empty boxes is inserted accordingly.
2. **Enter text.** A text string entered by a user replaces the cursor. Numerical data or variable identifiers can be entered in this way. A space character is interpreted as a metabox.
3. **Math keyboard.** (Leaf only) A 'virtual' keyboard containing special symbols is displayed. A selected character will replace the cursor symbol. (Note: any symbol in an expression is substituted by entries from the 'actual' keyboard whenever it is

aimed at by the pointer, that is, regardless of the cursor).

4. **Accent.** An selected accent symbol is attached above the cursor. The following types are available — dot, didot, tilde, bar, hat, check, acute, grave, breve, and vector.
5. **Delimiters.** A pair of delimiters whose size is adjusted to the cursor is generated. The following types are available — parenthesis, bracket, brace, angle bracket, floor, ceiling, vertical bar, double vertical bar, and root sign.
6. **Reduce to box.** The cursor is reduced to a metabox. The erased content is saved to the kill-buffer (to be described later). If the cursor is a row in a matrix, a corresponding row of box symbols are created instead; the erased content is saved as a row vector.
7. **Delete.** The cursor is removed from the equation, the logical structure of which is adjusted if necessary (such as when a superscript or a numerator is deleted). The cursor content is saved to the kill-buffer.
8. **Erase All.** The entire equation is deleted, and saved to the kill-buffer.

Template :

Mathematical templates are organized into six categories. The options 'user defined' through 'matrix' are used to select a template which will be plugged into the cursor symbol. The remaining commands help to manage these template libraries.

1. **User defined.** (Leaf only) This category contains templates which are created by the user. The library is displayed from which the user selects a template.
2. **Relation.** (Leaf only) This library holds system-provided templates which represent a mathematical relationship (such as equality, subset, tendency, and the like).
3. **Unary expression.** (Leaf only) This library holds system-provided templates which are unary expressions (such as trigonometric or logarithmic functions).
4. **Binary expression.** (Leaf only) This library holds system-provided templates which are binary in nature (such as addition or fraction).
5. **n-ary expression.** (Leaf only) This library holds system-provided templates which have an arbitrary number of subexpressions (such as summation or integral).
6. **Matrix.** (Leaf only) A grid of boxes appears. A user indicates the desired matrix dimension by enclosing the appropriate number of boxes anywhere on the grid (using a rubber box).

7. **Include.** (Non-leaf only) The cursor expression is saved to a specified template library. The user also indicates the new template's position within the library.
8. **Replace.** (Non-leaf only) The cursor expression replaces an existing template in a specified library.
9. **Discard.** An existing template from a specified library is removed permanently.
10. **Show library.** Templates available in the specified library are displayed.

Buffer :

Three distinct buffers are being maintained. The *cut buffer* holds contents which the user stores explicitly with the 'cut' or 'copy' command. The *kill buffer* holds contents which are removed in the last destructive command (namely, 'remove', 'replace', or 'erase all'). The *result buffer* holds an expression returned by the last 'evaluate' command (to be described later).

1. **Copy.** (Non-leaf only) The cursor expression is copied to the cut-buffer.
2. **Cut.** (Non-leaf only) The cursor expression is cut to the cut-buffer. The effects are similar to a 'copy' followed by a 'remove' command.
3. **Paste.** (Leaf only) The content from any of the three buffers can be plugged into the cursor symbol. (Note: the only situation when the target of a paste operation need not be a single symbol is when the cursor is a matrix row of empty boxes, provided the buffer holds a row-vector of the same number of elements.)
4. **View.** The content of any of the three buffers can be examined.
5. **Undo.** The last destructive commands (namely, 'remove', 'replace', or 'erase all') is undone. Only one stage of recovery is available. (Although deleted contents are saved automatically to the kill-buffer, this command is useful when the logical structure of an equation has been altered as well).

Manipulate :

1. **Evaluate.** (Non-leaf only) The cursor expression is evaluated. The result is saved to the result-buffer which is displayed as well. The 'math' attributes associated with the expression and all its descendents must be assigned explicitly (as described below), unless the expression is built entirely from system-provided templates which already

contain this information.

2. **Transpose.** (Non-leaf only) Provided the cursor is a matrix expression, it is replaced by its transposition. Since one limitation of the editor is the row-dominance in handling matrices, this option is useful when manipulation of matrix columns is required.

Attributes :

These functions alter the attributes associated with the cursor node.

1. **Math object.** This attribute indicates the logical class of an expression, such as integral, summation, product, difference, etc. A user assigns the object type of the cursor accordingly.
2. **Math field.** This attribute indicates the field of its parent to which an expression belongs, such as integrand, summand, upper bound, lower bound, left operand, etc. A user assigns the field type of the cursor accordingly.
3. **Binding mode.** This attribute defines how the immediate children of an expression are bound together physically. A user alters this attribute to, say, center an originally left-justified vertical list, or to change a subscript expression to a superscript directly (without any cut/paste or append operations).
4. **Auto scaling.** The formatter carries out automatic scaling at various levels of an equation. A user can disable the 'shrink' option to, say, override the automatic scaling down of a numerator expression.

C.2 Implemented Templates

Relations :

equality	$\square = \square$
inequality	$\square \neq \square$
less than	$\square < \square$
less/equal	$\square \leq \square$
greater than	$\square > \square$
greater/equal	$\square \geq \square$
approximation	$\square \approx \square$
similarity	$\square \sim \square$
identity	$\square \equiv \square$
proportionality	$\square \propto \square$
approach	$\square \rightarrow \square$
definition	$\square \stackrel{def}{=} \square$
subset	$\square \subset \square$
improper subset	$\square \subseteq \square$
superset	$\square \supset \square$
improper superset	$\square \supseteq \square$
member	$\square \in \square$

Unary expressions :

sine	$\sin \square$
cosine	$\cos \square$
tangent	$\tan \square$

inverse sine	$\sin^{-1} \square$
inverse cosine	$\cos^{-1} \square$
inverse tangent	$\tan^{-1} \square$
sinh	$\sinh \square$
cosh	$\cosh \square$
tanh	$\tanh \square$
inverse sinh	$\sinh^{-1} \square$
inverse cosh	$\cosh^{-1} \square$
inverse tanh	$\tanh^{-1} \square$
root	$\sqrt{\square}$
factorial	$\square !$
inverse	$\square^{-1}$
reciprocal	$\frac{1}{\square}$
negation	$- \square$
ceiling	$\lceil \square \rceil$
floor	$\lfloor \square \rfloor$
natural log	$\ln \square$
exponential	$e^{\square}$
logarithm (base 10)	$\log \square$
logical not	$\neg \square$

Binary expressions :

addition	$\square + \square$
subtraction	$\square - \square$
add/minus	$\square \pm \square$
multiplication	$\square \times \square$
division	$\square \div \square$

alternative division	$\square / \square$
alternative division	$\frac{\square}{\square}$
union	$\square \cup \square$
intersection	$\square \cap \square$
power	$\square^\square$
subscript	$\square_\square$
complex (orthogonal)	$\square + j\square$
complex (exponential)	$e^{\square + j\square}$
complex (polar)	$\square e^{j\square}$

n-ary expressions :

limit expression	$\lim_{\square \rightarrow \square} \square$
summation	$\sum_{\square = \square}^{\square} \square$
alternative summation	$\sum_{\square \leq \square \leq \square} \square$
alternative summation	$\sum_{\square} \square$
product	$\prod_{\square = \square}^{\square} \square$
alternative product	$\prod_{\square \leq \square \leq \square} \square$
alternative product	$\prod_{\square} \square$
definite integral	$\int_{\square}^{\square} \square d\square$
indefinite integral	$\int \square d\square$
partial derivative	$\frac{\partial^2 \square}{\partial \square^2}$

Matrix :

any dimensions under 10×10 $\left(\begin{array}{cc} \square & \square \\ \square & \square \end{array} \right)$

APPENDIX D

Architecture for Tables

D.1 Logical Model

<table>	::=	<field-dominant> <record-dominant>
<field-dominant>	::=	[<table heading>] + { <simple field> <composite field> }
<composite field>	::=	<composite field heading> + { <simple field> <composite field> }
<simple field>	::=	<simple field heading> + { <entry> }
<entry>	::=	<elementary object> (as defined in Appendix A)
<record-dominant>	::=	<headings> + <body>
<headings>	::=	[<table heading>] + <heading list>
<heading list>	::=	{ <simple field heading> <composite headings> }
<composite headings>	::=	<composite field heading> + <heading list>
<body>	::=	<record list>
<record list>	::=	{ <simple record> <composite record> }
<composite record>	::=	<composite record heading> + <record list>
<simple record>	::=	{ <entry> }
<table heading>	::=	<text>
<composite field heading>	::=	<text>
<simple field heading>	::=	<text>
<composite record heading>	::=	<text>
<spanning link>	::=	<entry> → <entry>
<data-sharing link>	::=	<entry> → <entry>

D.2 Formatting Rules

Table (<headings> + <body>)
 → V-concat (<heading> + <body>)

Headings (<table heading> + <heading list>)
 → V-concat (<table heading> + <heading list>)

Heading List ({ <simple field heading> | <composite headings> })
 → H-concat ({ <simple field heading> | <composite headings> })

Composite Headings (<composite field heading> + <heading list>)
 → V-concat (<composite field heading> + <heading list>)

Body ({ <simple record> | <composite record> })
 → V-concat ({ <simple record> | <composite record> })

Simple Record ({ <entry> })
 → H-concat ({ <entry> })

Composite Record (<composite record heading> + <record list>)
 → H-concat (<composite record heading> + <record list>)

Record List ({ <simple record> | <composite record> })
 → V-concat ({ <simple record> | <composite record> })

APPENDIX E

Typographical Glossary and Acronyms

Hyphenation	The splitting of words that overflow from a line during the typesetting process. In computer formatting, the decision of where to break a word is based on algorithmic means, supplemented by table lookup for exceptions. The objective is typically to avoid too many consecutive hyphenated lines or breaking a word at an improper place, as illustrated by 'the-rapist'.
Kerning	The adjustment of spacing between some pairs of characters which would otherwise look excessively separated when appearing next to each other, as illustrated by 'W' and 'A' in the unkered 'WANG'. This is usually essential for slanted and italic fonts.
Markup	The instructions to a typesetter (being a person) prepared on a typescript by a copy editor (a person). In computer typesetting, a markup language enables the description of a document's content (with embedded typographic control) to a formatting system, and is also called formatter language.
Orphan	The first line or lines of a paragraph being left at the bottom of a page by itself due to a page break. In computer typesetting, it is possible to adjust interline spacing dynamically in order to optimize the overall distribution of orphans. See also widow.
Pagination	This process assign the content of a document to individual pages, and involves decision on page breaking, allocation of floating materials, insertion of headers, footers, footnotes, and the like.

PDL	(Page Description Language). A low-level language, though often human readable, used to describe the appearance of a page incorporating text and graphics. Unlike a markup language, it describes already formatted pages, and serves the communication between composition systems and printing devices. For a dynamic PDL like Postscript, a page is generated during the 'execution' of the code.
Point	A printer's unit of measurement which corresponds approximately to $\frac{1}{72}$ of an inch.
RIP	(Raster Image Processor). A unit associated with a raster printing device (such as a laser printer or typesetter) that interprets source codes of a page description language and performs the necessary rasterization to generate the page image.
Widow	The last line or lines of a paragraph left on the top of a page by itself. See also orphan.
WYSIWIS	(What You See Is What I See). A colloquial name for a situation where a team collaborate through the use interconnected terminals. See also computer supported cooperative work (§2.3.3.3).
WYSIWYG	(What You See Is What You Get). A colloquial name for interactive document systems. See Section 2.2.1.1.

APPENDIX F

Statistical Font Caching

Here, we present a font caching scheme that exploits the statistical properties of English text. Although not part of our document model itself, it is an implementation issue and would be useful in applications which involve interactive display of high-resolution text. We first look at the problem we are addressing. Then follows a brief background on the statistical nature and prediction of English text. That forms the basis of the caching scheme which is presented next. Finally, we discuss the results of simulations using the scheme and others which do not exploit the statistics of English text.

F.1 Introduction

The work described here addresses the problem of the efficient use of memory in storing high-resolution or multiple bit-plane font templates for display purposes where lookahead is not possible, as in an interactive text display system. An ideal WYSIWYG document system imitates closely the image of the intended printed page. This requires the display of characters in multiple fonts at various point sizes. Increasingly, therefore, characters are being treated as complex graphical objects which must be coded in some way and generated on demand. A problem inherent with the WYSIWYG concept is that the resolution of current display devices is generally lower than printing devices. On the Xerox Star document system having a screen resolution of 72 dots per inch, for instance, a 10-point font

needs a magnified mode for text, and fonts below 8-point are hardly recognizable [Smith et al 1982a]. Similar problems are still encountered in recent systems like VORTEX [Chen et al 1986a].

The solutions to this problem all involve more memory. Using either magnification or higher display resolution increases the memory that would be required to hold the bitmap of a character. Assuming uniform aspect ratio, the size of a bitmap increases as the square of the resolution. An obvious approach is to generate bitmaps as each character is drawn — economical in memory but computationally expensive. Alternatively, work has been done on using anti-aliasing techniques to achieve better quality of displayed text at lower resolution [Warnock 1980, Wilkes & Wiseman 1982]. This approach is promising as the trend of integrating text and images implies that many document systems will eventually support gray-level displays.

A system is unlikely to be able to hold locally a useful set of typefaces at all sizes for all symbols at once. Here, we will study a caching approach to manage the memory allocated for typefaces. As in any similar scheme, there must be a criterion for the memory replacement decision. We exploit here the statistical property of English for this purpose as well as for managing the cache in general.

F.2 Statistics of English

The statistical nature and predictability of English text and the redundancy of letters were studied by Shannon as early as in the 1950's [Shannon 1951]. The statistics of English text have also been used successfully in the Viterbi and other syntax-directed text recognition algorithms [Forney 1973, Neuhoff 1975]. English

text has been treated as a random data source, specifically a finite-state, discrete-time Markov source. The order of the Markov model indicates the assumed span of influence among neighboring characters. For instance, for text modeled by a second-order Markov source, it is assumed that the outcome of a character is only influenced by the previous two characters. In our actual simulations we use the zeroth by to the second order statistics of the letters 'a' to 'z' only, and deal with other characters differently as will be seen. However, the general principles could apply to all characters.

In general, after a sequence of $N-1$ previous letters, the transition probability to the next (or the N^{th}) letter is calculated from the N -gram distributions in standard text for an $(N-1)^{th}$ order model. Taking the second-order case again as an example, the frequent occurrence of the trigram 'pso' relative to others starting with 'ps' implies that the digram is quite likely to be followed by an 'o' (with a conditional probability of 0.35). Higher-order N -gram statistics enable us to have better predictions since the N -gram entropy drops as N increases, resulting in higher redundancy (or less uncertainty) of the text [Shannon 1951]. The N -gram entropy is defined as the per-letter entropy due to inter-character statistics extending over N adjacent letters, and is thus a measure of the *predictability* of a letter given the last $N-1$ characters.

If prediction is based on the unigram table alone as in a zeroth-order model, one totally ignores the inter-letter influence and assumes that the outcome of a character is unaffected by the preceding context. With real text, however, this is not the usual case and a higher-order model should be used. However, the inter-letter influence tends to be more consistent within a word than when the N -gram is bridging over two words. For instance, the frequent occurrence of the trigram 'the' does not really imply that a word terminating with the letter 't' is likely to be

followed by another starting with 'he'. Inter-word dependency, however, does exist due to some common phrases. In general, it is adequate and practical to keep the order of the model below the average word length which is about 4.5 [Shannon 1951].

F.3 The Caching Scheme

We will first give an overall picture of the caching mechanism, describing our assumptions and the environment for which it is intended. Then we will have a closer look at the management of the font cache, namely, how the initial font work set is determined and how font faults (to be defined later) are handled.

F.3.1 An Overview

The cache management scheme is intended for situations where high-resolution representations of symbols are needed promptly to display an ordinary English text stream. It is assumed that lookahead information on the occurrence of symbols is not available, that is, a symbol is not known until it arises. Furthermore, the system can only accommodate the typefaces of a limited set of symbols at a time. In other words, our scheme is applicable to a viewing or printing system involving a single processor with memory constraints. Despite this general scope, the application we have aimed for is a WYSIWYG text editor.

High-definition font masters are usually defined in an encoded and compressed form. The template of a symbol is decoded, scaled and rasterized before being

displayed. A symbol is assumed to exist in either of two states, namely, a *compressed* state where processing is needed and a *ready* state where it can be displayed instantly. (It is of course possible to have a more complex hierarchy of states.) At any time, only some "high priority" symbols are allowed to be in a ready state, residing in a dynamic store called the *font cache*.

We now define *font fault* as a situation where a symbol is not available in the font cache when it occurs in the incoming text. That symbol must then be decoded and placed in the ready state, but only after a symbol in the cache is removed to make room for it. The problem is to assign a priority to symbols in the cache and to transfer symbols in and out of the cache dynamically depending on the input symbol sequence. This has an analogy with paging in a virtual memory system. Here, paging occurs between the encoded and decoded states of a symbol, instead of between two levels of the memory hierarchy.

The goal of the allocation scheme is to minimize the occurrence of font faults, taking into account the pattern of occurrence as well, based on some criteria discussed in Section 7.4.2. The statistics of English text are exploited in the overall management of the font cache. The overall arrangement of the cache is illustrated in Figure 7-1.

F.3.2 Initial Font Work Set

An initial work set for the font cache would be created either when the system starts up or when a user switches to a new font or point size. In the latter case, most of the existing symbols in the cache would be overwritten. The system might have the option of selecting a new font without the work set initialization; this is

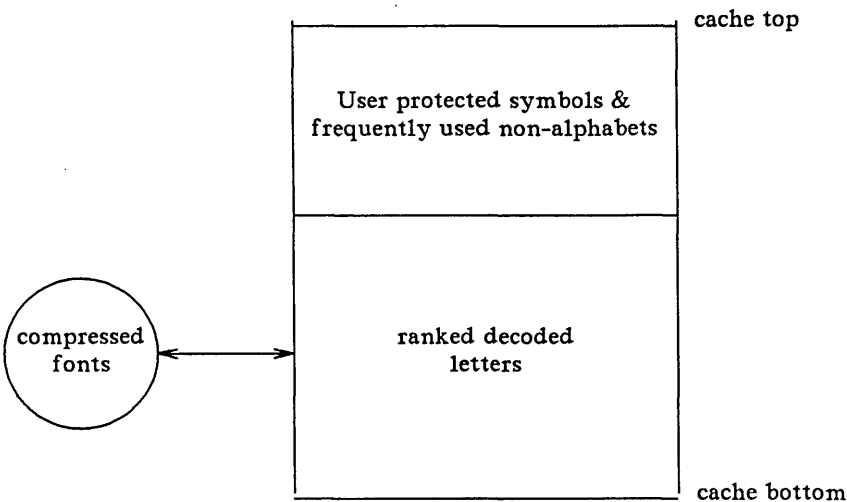


Figure F-1. The font cache

useful when the new font is needed briefly, such as for typing in a section heading.

Unigram statistics collected from standard English text are used to rank individual letters according to their frequencies of occurrence (in the order of e, t, a, o, i, n, etc.). Upper and lower case letters are distinguished in the cache as the two are not interchangeable for display purposes. Initially, all symbols in the font to be initialized are in the compressed state. Letters above a certain rank (depending on the cache size available) are decoded and scaled, and thus transfer to the ready state. In this way, the cache is preloaded with some generally frequent letters which at this point are ranked according to the unigram table.

However, as the text stream is being displayed, the cache ranking is adjusted according to the actual frequency of occurrence. When an already decoded symbol occurs in the input text, provided it is not at the cache top, its rank is swapped

with the letter immediately above it. This is intended to take care of any statistical discrepancy between the standard text from which the unigram table is derived and the one which is actually entered. Frequent symbols in the input text are therefore promoted and rare ones indirectly demoted. The former should tend to bubble to the cache top and therefore have a lower chance of being replaced when a fault occurs.

It is assumed that some commonly used non-alphabetical symbols always reside at the cache top and are protected, while rare ones are never brought into the cache even when they occur. This implies that one may wish to display a symbol without actually bringing it into the cache. In our simulations, the occurrence of non-alphabetical symbols does not cause any faults and therefore does not affect the cache allocation scheme. The system might also protect some selected symbols which, for some reasons, the user will be using frequently.

F.3.3 Font Faults

When an alphabetical symbol appears in the input text and is not available in the cache, a font fault occurs. It must then be decoded and brought into the cache. However, if there is no allocatable space in the cache, some symbol has to give up its cache space and return to the compressed state. Digram and trigram statistics are used at this point to select a suitable symbol for replacement. The aim is to avoid picking a symbol which may occur next in the input. A wrong replacement would result in two consecutive faults which is undesirable (as will be discussed in Section 7.4).

The digram table gives probabilities of ordered pairs of letters occurring. Similarly, the trigram table indicates the probabilities of various digrams giving rise to a subsequent letter. Entries in the two tables are of the following form :

digram 'xy' : $Pr(S_k=y/S_{k-1}=x)$

trigram 'xyz' : $Pr(S_k=z/S_{k-2,k-1}=xy)$

where S_k is the outcome of the k^{th} letter, and x, y, z can be any possible letters. The trigram transition probability matrix is illustrated in Figure 7-2.

To select a symbol for replacement, the font cache is searched from the character at the lowest rank. A symbol is tested to see if it is likely to appear next in the input stream, based on its transition probability given the previous context (using digram or trigram data as appropriate). Two approaches can be used at this point to select the victim. The letter with the minimum transition probability (or the 'least likely next') might be selected, requiring the entire cache to be searched. Alternatively, a letter could be replaced if its transition probability is below a certain threshold and, if not, the next symbol ranked above it is tested. This goes on until a replaceable letter is found, if there exists one. Both approaches (hereafter called hard and soft decisions) will be studied and, in the latter case, the effects of using various threshold values will be examined. A low threshold causes slow fault handling since many symbols may be protected unnecessarily, while using a high threshold symbols may be replaced even if they are fairly likely to occur next.

While the digram and trigram statistics are used in the replacement decision, the significance of unigram information should not be overlooked. In selecting a symbol for replacement, we are basically searching the cache whose ranking is initialized and constantly updated according to unigram statistics. The ranking

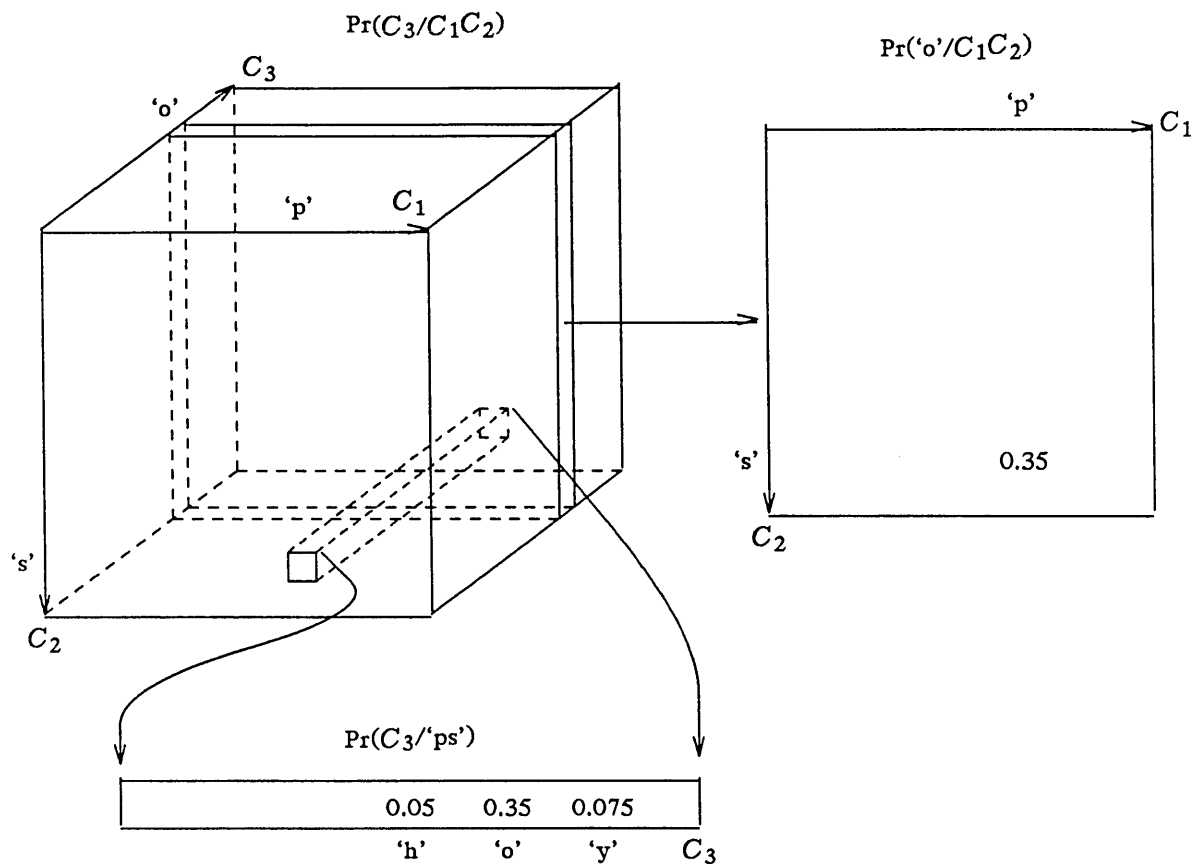


Figure F-2. Trigram probabilities illustrated by the sequence 'ps'

will reflect the actual frequency of usage; it is in discarding characters that we use statistical prediction.

Inter-word influence is here ignored. Also, for the sake of simplicity, we have assumed that the probability of occurrence of a symbol does not depend on its position within a word. The second letter of a word is predicted using the digram table, while letters in the third position onward are predicted with trigram data. For instance, consider the word "without". If the first letter 'w' causes a fault,

guessing an unlikely subsequent letter involves the digram probability column $\text{Pr}(x/'w')$, x being an arbitrary letter. Should 'h' cause a fault, the trigram column $\text{Pr}(x/'th')$ is used and this should avoid replacing the letter 'o' if it is already in the cache.

To appreciate the significance of prediction in the replacement decision, consider a situation where the letter 'q' arises, triggering a fault and that the letter 'u' is in the cache. If inter-letter influence is ignored and the selection for replacement is based on the unigram ranking alone, it is quite likely that 'u' will be victimized due to its generally rare occurrence. On the other hand, the digram table reveals that a 'q' is definitely followed by a 'u'. Had 'u' been replaced, an immediate subsequent fault would occur.

F.4 Cache Simulations

Unigram, digram and trigram statistics were computed from a sample of 150,000 words from the standard Lancaster-Oslo-Bergen (LOB) corpus of English text [Johansson et al 1978]. When collecting the statistics all non-alphabetical characters were treated as ordinary spaces, and interword digrams or trigrams were not counted. Furthermore, the distinction between upper and lower case letters is only drawn when preparing the unigram data, but not the digram or trigram tables. Therefore, the same trigram probability entry is used whether, say, the preceding text is 'th' or 'Th'. Our statistics are representative, but not definitive, particularly where rare but possible combinations are concerned.

F.4.1 The Experiments

We have conducted simulations on the caching scheme just presented as well as various other memory management strategies, using three cache sizes. Both the hard and soft decision variants of the statistical scheme have been studied (hereafter called STAT-H and STAT-S). Various threshold values have been used in the latter case. Four conventional replacement strategies have been included for comparison. These five replacement methods are summarized as follows.

- A. **Least Recently Used (LRU).** The symbol that has been idle or not displayed for the longest period of time is replaced.
- B. **Least Frequently Used (LFU).** The symbol with the least number of occurrences since it has been cached in last time is replaced.
- C. **First In First Out (FIFO).** The symbol which has stayed in the cache for the longest period is replaced, regardless of its occurrence.
- D. **Random (RND).** A symbol is picked at random for replacement. More specifically, the victim is the one being indexed by the remainder after dividing the fault's position in the input stream by the cache size.
- E. **Statistical (STAT).** The three main issues are restated below :
 - 1. An initial working set is created based on the unigram statistics; that is, generally frequent symbols are prefetched into the cache and are ranked accordingly. An area at the cache top is allocated for frequently used non-alphabetic and user selected symbols; characters in this area are never replaced.

2. An input symbol causing a fault is only placed in the cache if it is alphabetic. That letter is initially ranked such that all symbols above it in the cache have higher unigram probabilities (and thus generally more frequent). On the other hand, a letter which is already in the cache is promoted, that is, its rank is swapped with the one above it.
3. To select a symbol for replacement, the cache is searched starting from the bottom, and :
 - if hard decision (STAT-H) is used, the victim will be the letter with the minimum probability of being the next symbol (digram or trigram as appropriate). If, however, a letter with zero transition probability is encountered while the cache is being scanned, it is replaced anyway.
 - if soft decision (STAT-S) is used, the victim is the first letter found whose transition probability is below a threshold. Hence, the condition for replacement of a symbol x is as follows :

$$\text{digram : } Pr(S_{k+1}=x/S_k) < P_t$$

$$\text{trigram : } Pr(S_{k+1}=x/S_{k-1,k}) < P_t$$

where S_k denotes the current input symbol causing the fault. If a replaceable letter is not available, then the one with the least unigram probability is selected instead.

F.4.2 Evaluation Criteria

In any paging system, one objective is to minimize the overall number of faults. In

our application, a fault leads to the delayed appearance on screen of the symbol triggering the fault. It is therefore undesirable to have consecutive or clustered faults. The performance is here assessed using three measures.

- A. **Mean Distance Between Faults (MDBF).** This is the mean separation between successive font faults expressed in number of letters. It indicates the general frequency of faults that have occurred without information about the pattern of their occurrence.
- B. **Interfault gap distribution ($d_0, d_1, d_2 \dots$).** d_n is the percentage of the overall faults being separated by n letters (e.g. d_0 is the percentage of immediate faults). The short-gap distributions provide a measure of clustering of faults. We have no psychological data on interactive user response, but feel that successive faults as close as five characters apart will cause concern, if the delay is significant compared to the typing speed.
- C. **Bad caches.** A bad cache occurs when a letter that has been replaced is actually the next incoming one. This measure is the percentage of the total faults which result in bad caches. Bad caches are different from (and in fact, a subset of) d_0 introduced above. Where d_0 includes all immediate fault pairs, bad cache refers only to those for which the second fault is directly caused by a poor replacement in the first. This highlights the effect of the statistical replacement strategy whose purpose is to avoid bad caches.

Two allocation strategies may have the same MDBF but entirely different interfault gap distribution. Although the overall displaying or printing speed will be the same, the one with the least clustering of faults is preferred in an interactive system.

Method	MDBF	Bad Caches	d0	d1	d2	d3
LRU	3.41	3.88	35.60	24.37	15.16	8.44
LFU	1.91	3.48	72.22	20.69	3.46	0.74
FIFO	3.16	4.15	39.36	24.61	14.34	7.93
RND	3.14	3.24	40.33	25.10	14.13	7.47
STAT-H	4.60	0.12	18.75	23.88	19.17	10.27

cache size = 10 symbols

Method	MDBF	Bad Caches	d0	d1	d2	d3
LRU	5.77	2.55	19.74	18.62	14.75	9.57
LFU	2.68	2.82	48.72	26.94	12.89	4.65
FIFO	4.75	3.64	26.47	20.55	14.61	9.84
RND	4.76	3.00	26.43	20.90	14.59	9.51
STAT-H	6.58	0.04	13.43	19.55	16.42	9.35

cache size = 15 symbols

Method	MDBF	Bad Caches	d0	d1	d2	d3
LRU	11.07	1.29	12.56	12.69	10.93	7.33
LFU	4.59	2.13	23.30	23.71	17.39	10.52
FIFO	7.75	3.13	18.31	15.51	12.24	8.95
RND	7.17	3.04	20.22	15.75	12.25	8.81
STAT-H	11.22	0.04	8.37	13.10	12.09	7.36

cache size = 20 symbols

TABLE F-1. Results of simulations for various methods

F.4.3 Discussions

The results of the simulations using the STAT-H and the four non-statistical methods are presented in Table 7-1. The interfault gap distributions for these methods are depicted in Figure 7-3 for a cache size of 20. The results of STAT-S using various thresholds are found in Table 7-2, with the data of STAT-H and LRU included for comparison. The effects of the threshold value on the

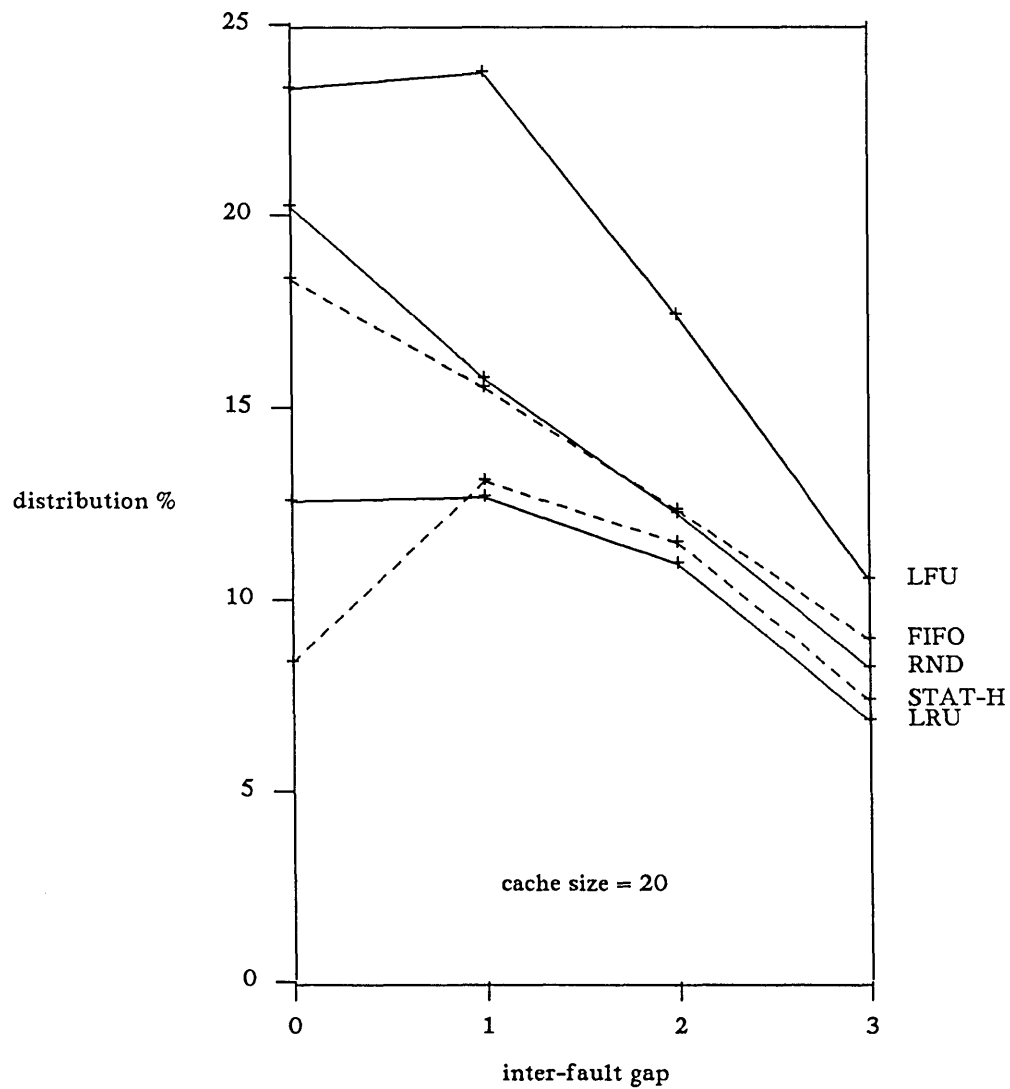


Figure F-3. Inter-fault gap distributions

performance of STAT-S is presented graphically in Figure 7-4 and 7-5.

From Table 7-1, one can notice that the LRU replacement method performs better than the other three conventional strategies. As far as bad caches is

Threshold	MDBF	Bad Caches	d0	d1	d2	d3
0.025	14.38	0.11	9.15	11.53	9.27	5.45
0.015	14.04	0.08	8.57	11.41	9.64	5.52
0.005	13.38	0.08	8.33	11.36	10.34	5.96
0.001	13.04	0.04	8.20	11.02	10.86	6.35
0.0005	13.89	0.09	8.37	10.90	10.24	5.82
0.0001	14.00	0.14	9.07	11.18	9.97	6.00
STAT-H	11.22	0.04	8.37	13.10	12.09	7.36
LRU	11.07	1.29	12.56	12.69	10.93	7.33

(STAT-H and LRU included for comparison)

TABLE F-2. Results of STAT-S using various thresholds (size = 20)

concerned, FIFO is the least impressive, while LFU is generally the worst in terms of MDBF and interfault gap distributions. The STAT method (hard or soft decision) works better than all these methods. In particular, STAT causes considerably fewer bad caches than the LRU method although the MDBF is not much different; that is exactly the purpose of using predictive replacement. Consequently, the percentage of immediate faults (d0, not bad caches) is relatively low as well.

We feel that d1 could also be suppressed further if we extend the existing replacement model to avoid discarding not just the next incoming letter, but the one after that as well. To achieve this we would need the conditional probabilities $Pr(S_{k+2}/S_{k-1,k})$ apart from $Pr(S_{k+1}/S_{k-1,k})$ which we are using now. This could be derived from quadgram data with which we could achieve a further stage of lookahead via prediction.

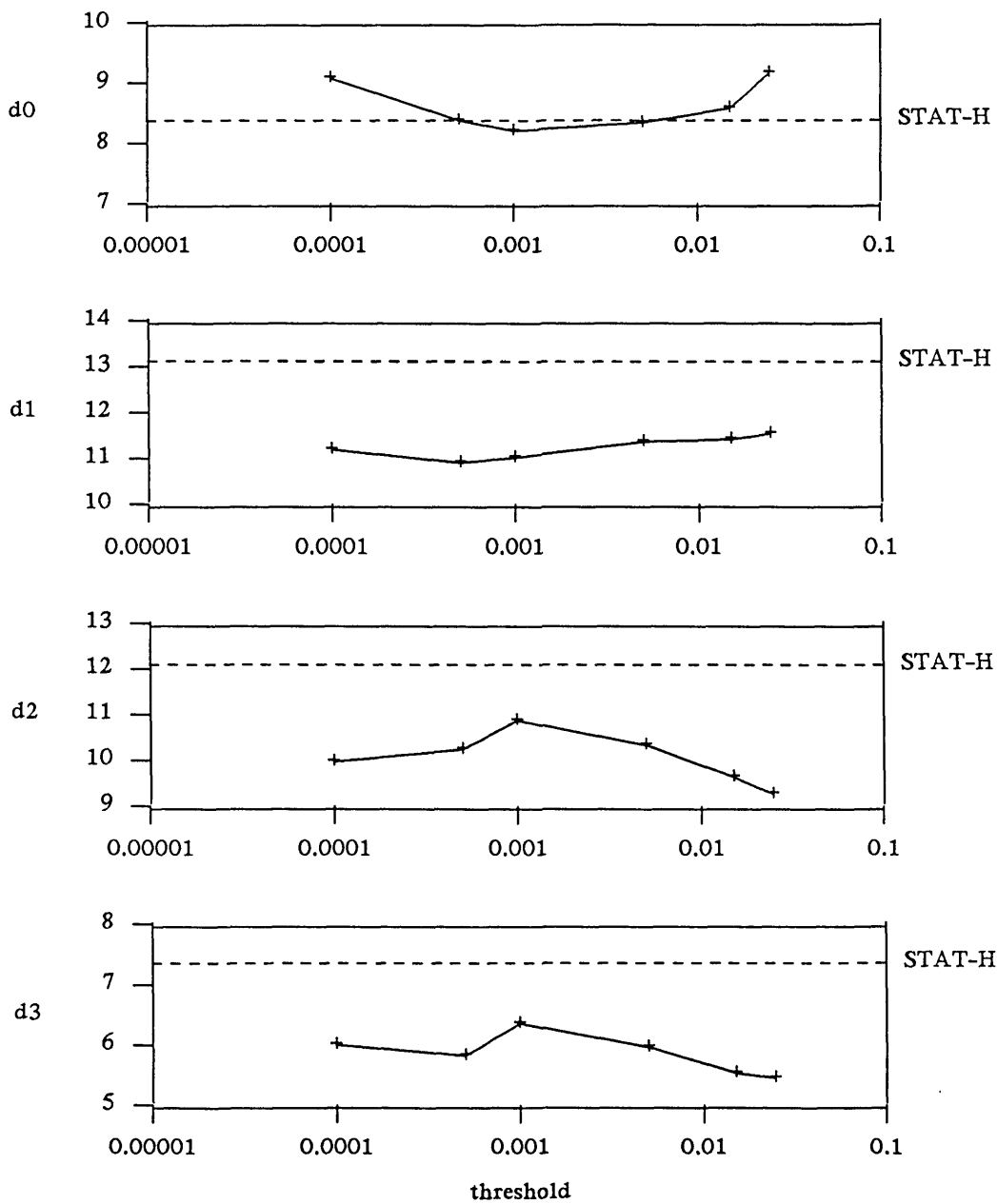


Figure F-4. Threshold versus inter-fault distributions for STAT-S (size = 20)

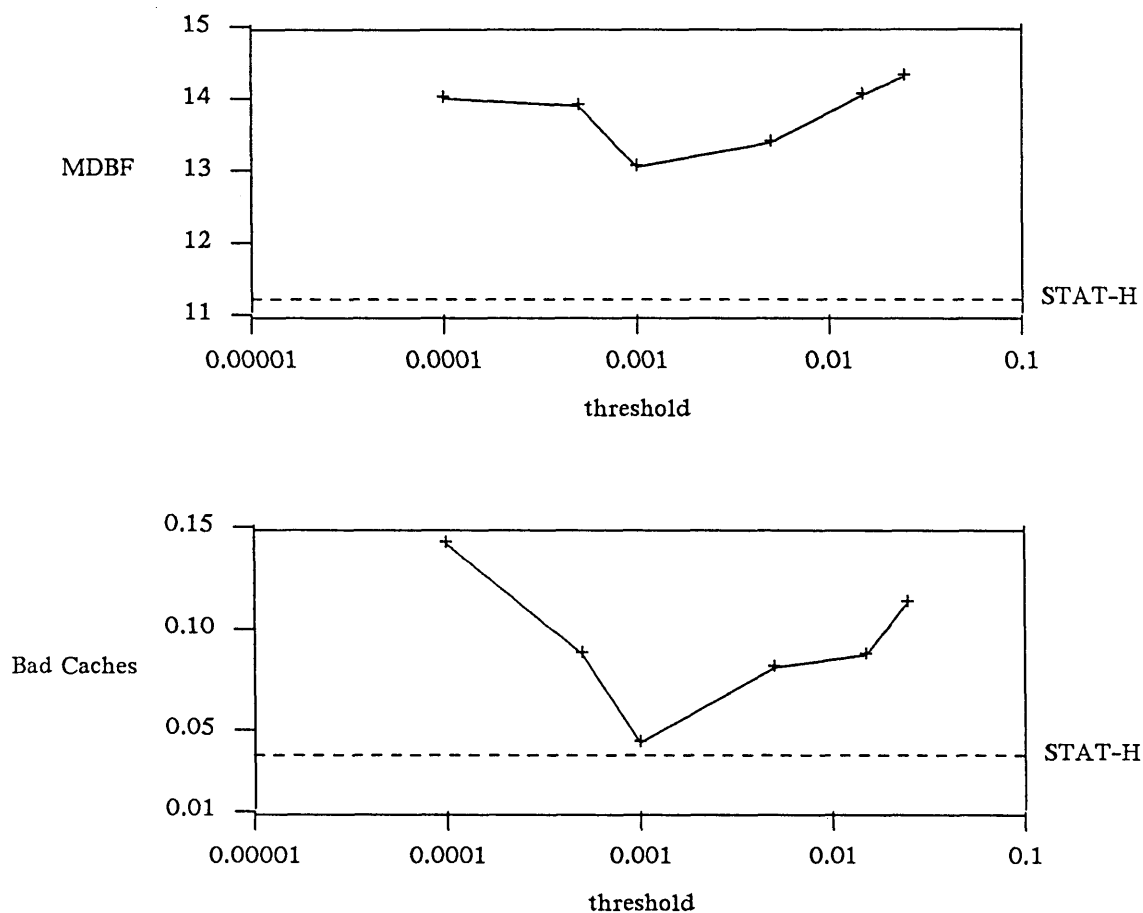


Figure F-5. Threshold versus MDBF and bad caches for STAT-S

The probability threshold used in the STAT-S method is also important. Using lower values give better performance, as we are then more stringent in the predictive replacement decision. It can be seen from Figure 7-4 and 7-5 that, when approaching from high thresholds, lower values render fewer bad caches, $d0$ and $d1$, which is quite natural. On the other hand, $d2$ and $d3$ actually increase, and the MDBF drops. Lower thresholds therefore tend to cause fewer immediate faults, though more overall faults result. However, as the threshold falls below a

certain range of values, the performance begins to degrade. This is because we are trying to only replace letters whose transition probability is below such a small value that no victim is available. Consequently, the unigram table alone is used most of the time, and the digram and trigram data are not exploited at all. Furthermore, the turning point in Figure 7-4 occurs where the threshold is somewhere just below 0.001. This is consistent with the fact that non-zero values below the order of 10^{-4} actually do not exist in the trigram transition probability table we used.

As far as bad caches and dO are concerned, the performance of STAT-H and STAT-S are comparable when the optimum threshold is used for the latter. On the other hand, STAT-H is actually inferior to STAT-S in terms of the MDBF (see Figure 7-5). This is because STAT-H always replaces the letter having the smallest transition probability, while STAT-S seeks the first acceptable letter when searching from the cache bottom. STAT-H will nearly always replace letters having a higher unigram rank than those STAT-S would choose — less likely to occur immediately but in general more frequent and so more likely to occur soon. Thus, STAT-H achieves fewer bad caches at the expense of more overall faults.

F.5 Conclusions

In this chapter, we have presented a font caching scheme whose performance is better than others which do not exploit the statistical nature of English text. The scheme is not well integrated with the rest of this thesis, which is on document architecture, but might be useful in an interactive high-resolution text editor with memory constraints.

A problem similar to the one addressed here, but on a larger scale, occurs in the real typesetting situation, where the local memory is insufficient to hold the rasterized font representations of all symbols at a time. In one example, the authors tackle this problem by managing a fast local cache using lookahead information on the incoming text stream [Fuchs & Knuth 1985]. Lookahead is used in both replacement and prefetch decisions. The algorithms have been proved to be optimal. A multiprocessing capability is assumed as characters are pre-rasterized while a current line is being typeset.

The work presented in this chapter is intended for situations in which lookahead on the input sequence is not available, such as an interactive environment. The replacement scheme used here predicts the next character rather than looking ahead. The fact that the statistical scheme works better than other replacement methods (including LRU which is used in at least one intelligent laser printer) suggests that the scheme could be useful in typesetting or other printing situations where a single processor has to deal with the incoming text serially.

The next chapter will summarize the work presented in this thesis, and will conclude with a discussion on the merits and shortcomings of an integrated, structure-oriented document system.